PathScale
High Performance Compilers

ENZO™

PathScale
High Performance Compilers

Table of Contents

1 Introduction

The PathScale ENZO™ Suite combines the HMPP (Hybrid Multicore Parallel Programming) open standard with direct code generation for NVIDIA® Tesla GPUs. This approach uses the strength of the GPU as a hardware accelerator (HWA) to replace traditional SIMD computing units.

Using HMPP directives with PathScale ENZO™ allows the programmer to write hardware independent applications where hardware-specific code is dissociated from the legacy code. Applications do not have to be explicitly rewritten for a target architecture.

Special thanks to CAPS Enterprise for giving us permission to reuse portions of their HMPP Workbench User Guide [R1] for the related notes, examples and HMPP directive syntax.

1.1 PathScale ENZO™ Overview

PathScale ENZO™ currently supports HMPP Fortran which, combined with the ENZO™ runtime, allows seamless execution of ENZO™ GPGPU applications. Future versions of ENZO™ will include support for HMPP C, C++ and ENZO™ C++ Templates.

To improve how quickly your application runs, ENZO™first identifies the regions of the application's source code that are suitable for the HWA target. Those regions then become regions or functions called “HMPP codelets” (see Section 2.1) using the HMPP directives. The hardware-accelerated versions of the regions or codelets are defined in the same source language as the rest of the program, such as Fortran, using the HMPP programming model.

The HMPP annotated source code is parsed by the PathScale Fortran frontend to translate the HMPP directives into calls to the ENZO™ runtime API (see Section 2.3). The ENZO™ runtime API is in charge of managing the concurrent execution of the codelets and regions.

HMPP directives also allow you to group codelets. Based on the codelet approach, these groups allow the programmer to use data already available on a hardware accelerator so that these data can be shared between different codelets executing at different times, without any additional data transfer between the host memory and the HWA.

Figure 1 shows how an ENZO™ application generates and compiles code. The native code and HWA code take the same path until the final stage when the compiler optimizes down to native heterogeneous assembly.

Figure 1 - PathScale ENZO™ Compilation Process

<insert image>

1.2 ENZO™ Runtime Overview

The ENZO™ runtime API controls the remote procedure calls to the HWA. Linked to the application, this library allocates memory and initializes the HWA to allow the execution of the codelets and regions. It relays communications between the host and the HWA and manages the asynchronous execution of regions and codelets.

1.3 PathScale ENZO™ Code Generation

PathScale ENZO™ generates direct-to-native HWA instruction code to maximize performance . The entire process is a unified solution, which does not rely on source-to-source conversion, and takes advantage of the PathScale HPC compute-focused NVIDIA® Tesla drivers.

Currently, ENZO™ only supports NVIDIA® Tesla C1060 and C1070 systems, but we intend to support Tesla 20xx by Q3 2010.

1.4 Scope of this Document

This manual covers the PathScale ENZO™ runtime, code generator and HMPP directives.

For documentation on the compiler CLI interface and installation instructions, please refer to the PathScale ENZO™ CLI User Guide and PathScale ENZO™ Installation notes.

2 HMPP Concept

HMPP is based on the concept of codelets, functions that can be remotely executed, and regions, which are areas of code meant to be executed on the target HWA. The ENZO™ runtime API library is in charge of remote procedure calls (RPCs) to the HWA, as well as managing the HWA's resources. HMPP directives can define a group of codelets, allowing the programmer to share data between different codelets that may run at different times on the HWA. We will refer to individual codelets as "stand-alone codelets" in the rest of the document to distinguish them from groups of codelets.

Please note that while PathScale ENZO™ does semantic checking on the directives, this does not guarantee that all errors of incorrect usage will be reported. Misuse of the HMPP directives may lead to erroneous results.

2.1 The HMPP Codelet Concept

A codelet is a computational part of a program located inside a function. It takes several scalars and array parameters, performs a computation on these data and returns the data. The result is passed by some parameters given by reference (`INTENT(inout)` in Fortran). The function does not support any return code (it is like a subroutine procedure in Fortran). The execution of a codelet is considered as atomic in that it does not have an identified intermediate state or data. The execution has no side effects.

Codelet parameters are classified into two types:

- Non-scalar parameters, which are restricted to array data types
- Scalar parameters, which are transferred by value

The transfer of non-scalar parameters is performed via the ENZO™ Runtime protocol. The size of all parameters must be known before the transfer of any parameter and before the codelet is executed.

A codelet has the following properties:

1. It is a pure function.
    - It does not contain static or volatile variable declarations nor refer to any global variables, unless these have been declared by an HMPP directive “resident” (see chapter 3.6.3 for more details).

- It does not contain any function calls with an invisible body (that cannot be inlined). This includes the use of libraries and system functions such as malloc, printf etc.
- Every function call must refer to a static pure function (no function pointers).

2. It does not return any value (void function in C or a subroutine in Fortran).
3. The number of arguments should be fixed (no variable number of arguments like vararg in C).
4. It is not recursive.
5. Its parameters are assumed to be non-aliased.
6. It does not contain callsite directives (RPC to another codelet) or other HMPP directives.

These properties ensure that a codelet RPC can be remotely executed by an HWA. This RPC and its associated data transfers can be asynchronous.

By default, all the parameters are uploaded to the HWA just before the RPC and downloaded just after it has finished executing.

The examples of code below will demonstrate the correct and incorrect ways to use and define codelets.

This is an example in C of a correctly-written codelet:

Listing 1 — Correct codelet definition

```
#pragma hmpp testlabel1 codelet, target=TESLA1, args[v1].io=out
static void codeletOk(int n, float v1[n], float v2[n], float v3[n]) {
  int i;
  for (i = 0 ; i < n ; i++) {
    v1[i] = v2[i] + v3[i];
  }
}
```

You cannot use global variables in the body of a codelet because the memory is not shared between the HWA and the CPU.

Listing 2 - Incorrect codelet definition due to use of a global variable

```
......
float globalVar[SIZE];
......
#pragma hmpp testlabel1 codelet, target=TESLA1, args[v1].io=out
static void codeletNotOk(int n, float v1[n], float v2[n], float v3[n]) {
  int i;
  for (i = 0 ; i < n ; i++) {
    v1[i] = v2[i] + v3[i]*globalVar[i];
  }
}
```

To fix the error, the global variable needs to be passed as a parameter to the codelet or be declared as a `resident` variable (see chapter 3.6.3).

You cannot use aliasing between parameters in a codelet. The following code produces an erroneous result due to the aliasing between `v1` and `v2` which point to the same caller parameters (see line 18, at the “callsite” level). On the device, the parameters are in independent data structures.

Listing 3 — Incorrect codelet definition due to aliasing between parameters

```
/* Legal codelet declaration */
#pragma hmpp testlabel1 codelet, target=TESLA1, args[v1].io=inout
  static void codeletNotOk(int n,
                           float v1[n],
                           float v2[n],
                           float v3[n]) {
  int i;
  for (i = 1 ; i < n ; i++) {
    v1[i] = v2[i-1] + v3[i];
  }
}
int main(int argc, char **argv) {
  ............
  /* wrong codelet use: the first two vectors are the same array */
#pragma hmpp testlabel1 callsite
  codeletNotOk(n, t1, t1, t3);
  ............
}
```

2.2 HMPP Codelet Remote Procedure Call and groups of codelets

The HMPP 1.5 directives standard specifies that the execution of a codelet should be atomic. By default, all input parameters were uploaded to the HWA before the RPC. The size of the parameters had to be known at runtime to initiate the memory transfers. They were provided either in the codelet's declaration (as explicit size of arrays in the prototype) or as parameters in the HMPP directives. The output parameters were downloaded back to the host memory once the codelet had successfully finished executing.

These rules still appear in the HMPP 2.0 directives standard, however the introduction of groups of codelets allows the programmer to execute several codelets as a sequence, sharing the same hardware, memory, and data. This approach reduces the overhead due to successive allocation and release of memory and hardware. It also reduces the data transfer overhead between the host memory and the HWA memory.

The management of the hardware accelerator is the same, except that it now remains allocated for the execution of the whole group (not just during the execution of each individual codelet as in version 1.5). This ensures that, once the data has been uploaded to the HWA accelerator, it's accessible to all the codelets in the same group.

Data management differs from 1.5 since it is necessary to manage the data throughout the application for different codelets in the same group.

Before loading and executing a codelet or a group of codelets on an HWA, the ENZO™ runtime ensures thatThe HWA is present and available (i.e. not busy) in the platform and an HWA implementation of the codelet or the group of codelets is available.

Unless all those conditions are satisfied, ENZO™ will either wait for the HWA to be available or not run.

2.2.1 Execution Error with Asynchronous or Synchronous Codelet RPCs

In the case of a synchronous (default) or asynchronous codelet RPC, when an error occurs ENZO™ will call `abort()`, report the error and exit.

Asynchronous data transfer and asynchronous codelet execution are hardware accelerator-dependent.

2.3 ENZO™ Runtime API Library Routines

The ENZO™ runtime API manages the concurrent execution of HWA implementations of the codelets and regions, in combination with native code.

2.4 ENZO™ Memory Model

In the current version of ENZO™, the memory addresses managed at the host level and at the HWA level are different (see Figure 3). The “application” and the ENZO™ runtime API have their own private memory. ENZO™ deals with this in a way transparent to the user. ENZO™ is the programming "glue" between target-specific programming environments and general-purpose programming.

Figure 3 - ENZO™ memory model

<insert image>

3 HMPP Directives

3.1 Introduction

The HMPP 2.0 directives are metadata added in the application's source code. They are safe as they do not change the original code. They address the remote execution (RPC) of a function as well as the transfers of data to/from the HWA memory.

The simplest use of HMPP directives is two directives made of a codelet declaration and a callsite marker. They are identified by a unique label given in each directive. The scope of the label is the compilation unit but the label must be unique for the whole application. For instance, in the listing below the directive at line 2, testlabel declares a TESLA1 codelet implementation to be run on an NVIDIA® GPU. The call to this codelet is on line 31.

It should be noted that the HWA implementation of a codelet is specific to a call site. This is because the use of an HWA is specific to both a computation and its context.

Listing 4- HMPP codelet source code example

```
....
#pragma hmpp testlabel codelet, target=TESLA1, args[vout].io=inout
static void kernel(unsigned int N, unsigned int M,
                         float vout[N][M], float vin[N][M]){
  int i, j;
  for(i = 2; i < (N-2); i++) {
     for(j = 2; j < (M-2); j++) {
       float temp;
       temp = vin[i][j]
          + 0.3f *(vin[i-1][j-1] + vin[i+1][j+1])
          - 0.506f *(vin[i-2][j-2] + vin[i+2][j+2]);
        vout[i][j] = temp * (vout[i][j]);
      }
    }
 }
 int main(int argc, char **argv){
   unsigned int n = 100;
   unsigned int m = 20;
   int i, j;
   float resultat = 0.0f;
   float out[n][m];
   float in[n][m];
   ...
   // init
   for(i = 0 ; i < n ; i++){
      for(j = 0 ; j < m ; j++){
        in[i][j] = (COEFF) * (-1.0f);
        out[i][j] = (COEFF) + (j * 0.01f) ;
      }
   }
 #pragma hmpp testlabel callsite
     kernel(n,m,out,in);
     ....
     printf("result : %f\n",resultat);
}
```

Table 1 shows the HMPP directives. These directives address different needs: some of them are dedicated to declarations, others to managing the execution of the codelet.

Table 1- HMPP Directives

	Control flow instructions	Directives for data management
Declarations	• codelet • group	• resident • map • mapbyname
Operational Directives	• callsite • synchronize • region	• allocate • release • advancedload • delegatedstore

3.2 Concept of set of directives

The concept of directives and their associated labels allow you to recreate a coherent structure on a whole set of directives spread throughout an application. This is a fundamental part of HMPP.

There are two kinds of labels:

- Directives associated with a codelet. In general, the directives carrying this kind of label are limited to managing only stand-alone codelets.
- Directives associated with a group of codelets. These labels are written as follows: “<LabelOfGroup>“, where “LabelOfGroup” is a name specified by the user. In general, the directives which have a label of this type relate to the whole group.

The concept of a group is reserved to a class of problems which requires a specific management of the data throughout the application to obtain performance. In the following, for each directive, we will present both notations for:

- A stand-alone codelet context: only one set of directives associated to one codelet is defined. Note that in an application, several separate set of directives can be defined.
- A group of codelets: the set of directives deals with the definition of several codelets in the same group.

The HMPP directives with different labels do not see each other, i.e. a directive of a given label does not interfere with a directive using a different label.

Please note that inside a set, directives can only interact by sharing data and data cannot be shared between two distinct sets of directives.

3.3 Syntax of the HMPP directives

In order to simplify the notation, regular expressions will be used to describe the syntax of the HMPP directives. A summary follows.

- “?” A question mark indicates there is either no preceding item or one preceding item.
- “*” An asterisk indicates there are zero or more instances of the preceding items.
- “+” A plus sign indicates there are one or more instances of the preceding items.

To keep the notation simple , we use the same notation for stand-alone codelets and groups of codelets. The main difference between the two syntaxes is an additional label to manage the groups.

We also have a color key for describing syntax directives:

- Reserved HMPP keywords are in **blue**
- Elements of grammar which can be declared as HMPP keywords are in **red**
- Code which is meant to be empahsized is in **bold black**
- Highlighted code is in **magenta**

In stand-alone codelet context, the general syntax of the HMPP directives is (for C and C++):

```
#pragma hmpp codelet_label directive_type [, directive_parameters]* [&]
```

- The syntax for Fortran 95, 2003 and 2008 is:

```
!$hmpp codelet_label directive_type [, directive_parameters]* [&]
```

Where:

- "<grp_label>" is a unique identifier naming a group of codelets. In cases where no groups are defined in the application, this label should be left out. A legal label name must follow this grammar: [a-z,A-Z,_][a-z,A-Z,0-9,_]*. Note that the "< >" characters belong to the syntax and are mandatory for this kind of label.
- "codelet_label" is a unique identifier naming a stand-alone codelet. A legal label name must follow this grammar: [a-z,A-Z,_][a-z,A-Z,0-9,_]*.
- "directive_type" is the directive's type.
- "directive_parameters" designates some parameters associated with the directive_type. These parameters may be of different kinds and specify either arguments given to the directive or a mode of execution (asynchronous versus synchronous for example).
- & and \ are used to continue the directive on the next line (same for C, C++ and Fortran).

Example of a simple codelet declaration with no group definition

```
#pragma hmpp codelet_label codelet, &
#pragma hmpp codelet_label directive_parameter &
#pragma hmpp codelet_label [, directive_parameter]*
```

Example of a codelet declaration inside a group

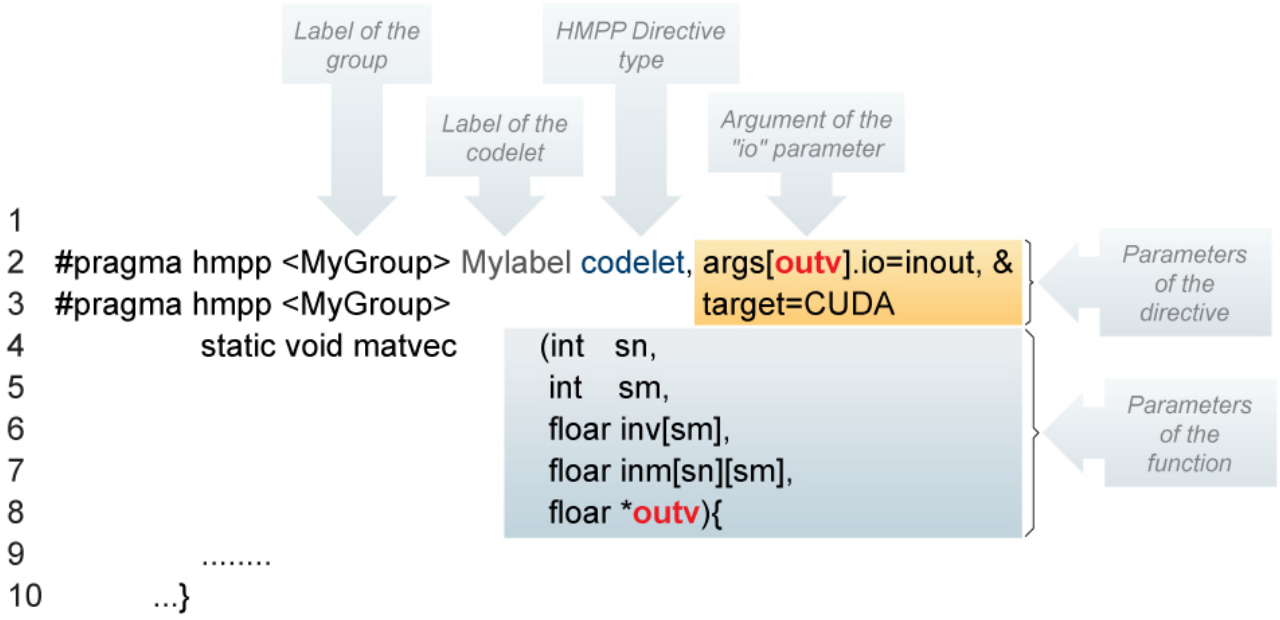
```
#pragma hmpp <grp_label> codelet_label codelet, &
#pragma hmpp <grp_label>               directive_parameter &
#pragma hmpp <grp_label>               [, directive_parameter]*
```

Furthermore, the directive's parameters may accept arguments.

We will define these two notions as follows:

- Parameters.These are parameters of directives,
- Arguments. These are arguments belonging to parameters.

Figure 4 - Description of parameters and arguments



In this example, `outv` *is a value of the directive parameter and points to the user's function arguments.*

Values of the directive parameters can be specified by: their formal name, their order in the function definition, or a range (in case several arguments need to be provided to the directive).

Example:

```
#pragma hmpp <grp_label> directive_type, args[arg_items].xxx
```

Where `args[arg_items].xxx` represents the directive parameter with

```
arg_items:        arg_item [ „;“ arg_item ]*
arg_item:         IDENTIFIER | NUMBER | arg_range | param_with_ident
arg_range:        NUMBER „-„ NUMBER
param_with_ident: ident „::“ [* | IDENTIFIER]
ident:            codelet_label | *
```

Where

- `IDENTIFIER` is the name of a parameter in the codelet prototype;
- `NUMBER` is the numerical position of a function's argument - starting from 0.

Listing 5 gives an example where

- `args[0-1]` point out `sn` and `sm`, respectively
- `args[inv]` designates `inv`
- `args[3]` designates `inm` and so on

Listing 5 - Directive parameters and arguments (stand-alone codelet notation)

```
#pragma hmpp simple1 codelet, args[0-1;inv].io=in, &
#pragma hmpp simple1          args[3].io=in,       &
#pragma hmpp simple1          args[outv].io=inout, &
#pragma hmpp simple1          target=TESLA1
static void matvec(int sn, int sm,
                   float inv[sm], float inm[sn][sm],
                   float *outv){
  ........
}
```

The following construction is also legal:

```
#pragma hmpp <MyGroup> delegatedstore, args[*::var_b]
```

The `delegatedstore` directive is applied on all the variables `var_b` defined in the group `MyGroup` (codelet parameters and resident variables if any).

Example:

```
#pragma hmpp <MyGroup> delegatedstore, args[::MyResVarData;cod1::var_a;*::var_b]
```

The `delegatedstore` directive is applied on the group `MyGroup` on the following variables:

- the resident data `MyResidentVarData`;
- the `var_a` argument of the codelet `cod1`;
- all the arguments called `var_b` defined in the group `MyGroup`

Please note that when many parameters of the same codelet are referenced, the following notation is also supported:

```
#pragma hmpp <MyGroup> delegatedstore, args[cod1::var_a;cod1::var_b]
```

This is equivalent to:

```
#pragma hmpp <MyGroup> cod1 delegatedstore, args[var_a;var_b]
```

The codelet label `cod1` has been moved to the beginning of the directive and removed from the variable declarations to shorten the directive.

Table 2 - summarizes the different way to access to the argumentsaccording to their scope

	By name	By rank (start from 0)	By range	All
Implicit current scope	MyArgument	3	0-5	*
Explicit codelet scope	MyCodelet::MyArgument	MyCodelet::3	MyCodelet::0-7	MyCodelet::*
Explicit resident scope	::MyResidentVariable			::*
Global scope	*::MyVariable			*::*

In the rest of this document, we will give most of our examples of directives in C. Fortran directives only differ by their prefix.

In C, C++ and Fortran the directives are not case sensitive.

3.4 Directives for Implementing the Remote Procedure Call on an HWA

Using an HWA involves a remote procedure call. A set of directives controls the implementation of the RPC:

1. The `codelet` directive marks a function as a codelet with the properties of its parameters (inputs and outputs).
2. The `callsite` directive declares the call to the codelet that is remotely executed.

3.4.1 codelet directive

A `codelet` directive requires that the function following it is optimized for a given hardware. Its label must not be used for anything else in the application.

A `codelet` directive must have a label. A group label is not required if no group is defined.

The codelet directive must be inserted immediately before the function is declared.

For a stand-alone codelet, the directive is:

```
#pragma hmpp codelet_label codelet [, version = major.minor[.micro]?]?
                           [, args[arg_items].io=[in|out|inout]]*
                           [, args[arg_items].size={dimsize[,dimsize]*}]*
                           [, args[arg_items].const=true]*
                           [, cond = "expr"]
                           [, target=target_name[:target_name]*]
```

For a group of codelets, the directive is:

```
#pragma hmpp <grp_label> codelet_label codelet [,version = major.minor[.micro]?]?
                             [, args[arg_items].io=[in|out|inout]]*
                             [, args[arg_items].size={dimsize[,dimsize]*}]*
                             [, args[arg_items].const=true]*
                             [, cond = "expr"]
                             [, target=target_name[:target_name]*]
```

Where:

- `<grp_label>` is a unique identifier associated with all the directives that belong to the group (definition and use).
- `codelet_label` is a unique identifier associated with all the directives that belong to the same codelet execution (definition and use).
- `version = major.minor[.micro]?` specifies the version of the HMPP directives to be considered by the preprocessor (for each of them, value may be positive or `null`).
- `args[arg_items].size={dimsize[,dimsize]*}` specifies the size of a non-scalar parameter (an array). Each dimsize provides the size for one dimension. dimsize must be a simple expression depending only on the scalar arguments of the codelets.
- `args[arg_items].const=true` indicates that the argument is to be uploaded only once. Note that even if there is only one codelet callsite associated with a codelet declaration, there can be several calls to the codelet, for example if the callsite is inside a loop.
- `args[arg_items1].io=[in|out|inout]` indicates that the specified function arguments are either `input`, `output` or both. By default, for codelets and resident, unqualified arguments are inputs. For HMPP regions, arguments are `INOUT`. The specification for this parameter drives the data transfers between the host and the HWA. Furthermore, it allows additional checks about how the data is used in HMPP applications.

Table 3 - Intent in Fortran versus HMPP Input/Output parameter policy

INTENT HMPP IO	Default	IN	OUT	INOUT
Unset	IN	IN	OUT	INOUT
IN	IN	IN	Error	Warning
OUT	OUT	Error	OUT	Warning
INOUT	INOUT	Error	Error	INOUT

In Fortran, the ".io" parameter can be omitted when an "INTENT" attribute is explicitly specified in the source code.

Table 4 – C language parameter versus HMPP Input/Output parameter policy

C Parameters HMPP IO	By Value	By Const address	By address
Unset	IN	IN	IN
IN	IN	IN	IN
OUT	Error	Error	OUT
INOUT	Error	Error	INOUT

In C, a scalar argument is passed by value, so its HMPP input/output property cannot be OUT or INOUT. A pointer argument with a const attribute has the same restriction (see Table 4).

- `cond = "expr"` specifies an execution condition as a boolean C or Fortran logical expression that needs to be true before the codelet will run. The expression must be correct and evaluate in all operational directive contexts (see Table 1). cond is useful to control when directives are executed. All directives are executed normally but they will still be executed even if, for example, a goto statement in the host code implicitly skips an HMPP directive. The host code is required to set up the expression `expr` so that if it wants to skip an HMPP directive `expr` evaluates to `FALSE`.
- `target=target_name[:target_name]*` specifies one or more targets for which the codelet must be generated. It means that according to the target specified, if the hardware is available *and* the codelet implementation for that hardware is also available, it will be executed. Otherwise, the next target in the list will be tried.

  The values of the targets can be one of the following:

  - TESLA1 for NVIDIA® Tesla C1060 and C1070
  - TESLA2 for NVIDIA® Tesla C20xx

For more information on the targets, please refer to section 7.

Listing 6 – Simple codelet declaration

```
#pragma hmpp simple1 codelet, args[outv].io=inout, target=TESLA1
static void matvec(int sn, int sm,
                   float inv[sm], float inm[sn][sm], float *outv){
  int i, j;
  for (i = 0 ; i < sm ; i++) {
    float temp = outv[i];
    for (j = 0 ; j < sn ; j++) {
      temp += inv[j] * inm[i][ j];
  }
  outv[i] = temp;
}
int main(int argc, char **argv) {
  int n;
  ........
#pragma hmpp simple1 callsite, args[outv].size={n}
  matvec(n, m, myinc, inm, myoutv);
  ........
}
```

Listing 7– codelet declaration inside a group

```
#pragma hmpp <myGroup> simple1 codelet, args[outv].io=inout, target=TESLA1
static void matvec(int sn, int sm,
                   float inv[sm], float inm[sn][sm], float *outv){
  int i, j;
  for (i = 0 ; i < sm ; i++) {
    float temp = outv[i];
    for (j = 0 ; j < sn ; j++) {
      temp += inv[j] * inm[i][ j];
  }
  outv[i] = temp;
}
int main(int argc, char **argv) {
  int n;
  ........
#pragma hmpp <myGroup> simple1 callsite, args[outv].size={n}
  matvec(n, m, myinc, inm, myoutv);
  ........
}
```

More than one `codelet` directive can be added to a function to specify different uses or execution contexts. However, there can be only one `codelet` directive for a given callsite label. An example appears below:

Listing 8 – Multiple codelet declarations (for stand-alone codelets)

```
#pragma hmpp simple1 codelet, args[outv].io=inout, &
#pragma hmpp simple1           cond ="n==1024", target=TESLA1
#pragma hmpp simple2 codelet, args[outv].io=inout, &
#pragma hmpp simple2           cond ="n==1024", target=TESLA1
static void matvec(int sn, int sm,
                   float inv[sm], float inm[sn][sm], float *outv){
  int i, j;
  for (i = 0 ; i < sm ; i++) {
     float temp = outv[i];
    for (j = 0 ; j < sn ; j++) {
      temp += inv[j] * inm[i][ j];
    }
    outv[i] = temp;
  }
}
int main(int argc, char **argv) {
  int n;
  ........
#pragma hmpp simple1 callsite, args[outv].size={n}
  matvec(n, m, myinc0, inm, myoutv0);
#pragma hmpp simple2 callsite, args[outv].size={n}
  matvec(n, m, myinc1, inm, myoutv1);
  ........
#pragma hmpp simple1 release
#pragma hmpp simple2 release
}
```

Note that if more than one `callsite` directive precedes a function call, only one of them can initiate an RPC call. The execution policy is based on the order of the callsite directives: the directives are evaluated one after the other. Thus, a callsite can only be launched if the condition of all previous `callsite` directives has failed *and* the condition of the current directive is true *and* the HWA is available. Subsequent directives will be ignored once one has been executed.

3.4.2 group directive

The `group` directive allows the declaration of a group of codelets. The parameters defined in this directive are applied to all codelets belonging to the group.

The syntax of the directive is:

```
#pragma hmpp <grp_label> group, [version = <major>.<minor>[.<micro>]?]? &
                               ,[target= target_name[:target_name]*]]? &
                               ,[cond = “expr”]?
```

Where the directive parameters are:

- `<grp_label>` a unique identifier associated with all the directives that belong to the group (definition and use). This label will have to be reused to run any codelet within a group.
- `version = major.minor[.micro]?` specifies the version of the HMPP directives to be considered by the preprocessor.
- `cond = "expr"` specifies an execution condition as a boolean C or Fortran logical expression that must be true to start the execution of the group of codelets. If a condition for a group is specified at this level, it will overwrite the existing codelet conditions. See the comments under the codelet directive for alternate applications of this cond parameter.
- `target=target_name[:target_name]*` specifies which targets to use and their order. MIf the corresponding hardware and codelet implementations for the specified target are available it will be executed. Otherwise, the next target specified in the list will be checked. For more information on targets, please refer to section 7 .

3.4.3 The callsite directive

The `callsite` directive specifies the use of a codelet at a given point in the program. Related data transfers and synchronization points that are inserted elsewhere in the application have to use the same label.

A codelet label is mandatory for the callsite directive. A group label is also required if the codelet belongs to a group.

The callsite directive must be inserted immediately before the function call.

The syntax of the directive for stand-alone codelets is:

```
#pragma hmpp codelet_label callsite
                           [, asynchronous]?
                           [, args[arg_items].size={dimsize[,dimsize]*}]*
                           [, args[arg_items].advancedload=[true|false]]*
                           [, args[arg_items].addr="expr"]*
                           [, args[arg_items].noupdate=true]*
```

For a group of codelets, the syntax is:

```
#pragma hmpp <grp_label> codelet_label callsite
                           [, asynchronous]?
                           [, args[arg_items].size={dimsize[,dimsize]*}]*
                           [, args[arg_items].advancedload=[true|false]]*
                           [, args[arg_items].addr="expr"]*
                           [, args[arg_items].noupdate=true]*
```

Where the directive parameters are:

- `<grp_label>` is a unique identifier associated with all the directives belonging to the group (definition and use).
- `codelet_label` is a unique identifier associated with all the directives belonging to the same codelet execution (definition and use).
- `asynchronous` specifies that the codelet execution is not blocking (default is synchronous). In asynchronous mode, all the output parameters have to be downloaded using the delegatedstore directive (see Section 3.5.2). A synchronize directive is mandatory before the first delegatedstore directive to insure that the codelet executes properly.

  When an asynchronous codelet is declared, a release directive is also mandatory (see section 3.4.5).

- `args[arg_items].size={dimsize[,dimsize]*}` specifies the size of a non-scalar parameter (i.e. an array) if it is not provided by the codelet prototype. Each dimsize provides the size for one dimension. The set is evaluated at runtime by an allocate directive, or by all callsite and advancedload directives within the group.
- `args[arg_items].advancedload=true` indicates that the specified parameters are preloaded (see Section 3.5.1). In this case, at the callsite directive level, HMPP will not load the specified data. Only in or inout parameters can be preloaded.
- `args[arg_items].addr="expr"` gives the address of the data to load, store or both.
- `args[arg_items].noupdate=true` this property specifies that the data is already available on the HWA so no transfer is needed. The user is responsible for making sure these data are actually on the HWA.

You can see examples of the `callsite` directive in Listing 8. If the condition of the directive is not `true`, or if no resources are available on the HWA, the native codelet code is used instead.

It should be noted that if there are no `allocate` and `release` directives (see chapters 3.4.5 and 3.4.5 respectively) in the directive set, the callsite directive will perform device acquisition as well as allocate the parameters then free them and release the device.

3.4.4 The synchronize directive

The synchronize directive specifies to wait until the completion of an asynchronous callsite execution.

As with the callsite directive, a label for the codelet is mandatory, as is a group label, if it belongs to a group.

The syntax of the synchronize directive for stand-alone codelets is:
:

```
#pragma hmpp codelet_label synchronize
```

For a group of codelets, the syntax is:

```
#pragma hmpp <grp_label> codelet_label synchronize
```

Where the directive's parameters are:

- <grp_label>: a unique identifier associated with all the directives belonging to the group (definition and use).
- codelet_label: a unique identifier associated with all the directives belonging to the same codelet execution (definition and use).

When the synchronize directive is used, the corresponding callsite directive must be set as asynchronous. If not, the directive will be ignored and a warning message will appear during compilation. If a synchronization point is encountered before the codelet is called, an execution error will occur.

Note that the synchronize directive is only a synchronization barrier. delegatedstore directives should follow if output data need to be downloaded from the HWA to the host.

3.4.5 The allocate directive

An HWA may need some time to be allocated or initialized before being used by a set of directives. Pre-allocating the hardware, before the RPC is called or any data uploaded, may improve execution time ,. This pre-allocation should be done via the allocate directive.

When an allocate directive is used, it must be executed before all other i directives.

To allocate memory in the HWA, ENZO™ evaluates the sizes of the non-scalar parameters during the execution either from the codelet or directly from an expression given by the user in the call site (see parameter size of the HMPP callsite directive, chapter 3.4.2). Note that once the size has been evaluated, it cannot be changed during any execution of the codelet up to the next release directive.

The syntax for stand-alone codelets is:

```
#pragma hmpp codelet_label allocate [,args[arg_items].size={dimsize[,dimsize]*}]*
```

IFor a group of codelets, the syntax is:

```
#pragma hmpp allocate [,args[arg_items].size={dimsize[,dimsize]*}]*
```

Where the directive parameters are:

- `<grp_label>` a unique identifier associated with all the directives belonging to the group (definition and use).
- `codelet_label` a unique identifier associated with all the directives belonging to the same codelet execution (definition and use).
- `args[arg_items2].size={dimsize[,dimsize]*}` gives an alternate way to evaluate the size of non-scalar codelet arguments. Each dimsize provides the size for one dimension. `dimsize` is an expression evaluable at the location of the directive (can be a variable, a value, an expression to evaluate etc.)

This directive is used when the callsite specifies an unknown size in the advancedload directive. The size must be specified for each dimension of the argument. Listing 9 illustrates the size declaration for two n-by-m matrices `inm` and `outv`.

Please, note that once a ".size" parameter is specified for an argument in an allocate directive, this value cannot be changed in an advancedload or delegatedstore directive.

The `allocate` directive is used for both asynchronous and synchronous RPCs. When used, the allocation step is not performed by other directives. This directive must therefore override the default in all other directives belonging to the same codelet.

Listing 9 - allocate directive example (for stand-alone codelets)

```
#pragma hmpp matvec allocate, args[inm;outv].size={n,m}
  ....
  while (...){
#pragma hmpp matvec callsite, asynchronous
    matvec(n, m, (inc+(k*n)), inm, (outv+(k*m)));
    ....
#pragma hmpp matvec synchronize
#pragma hmpp matvec delegatedstore, args[outv]
  }/* endwhile */
#pragma hmpp matvec release
```

3.4.6 The release Directive

The `release` directive specifies when to `release` the HWA for a group or a stand-alone codelet (this directive is generally used in association with the `allocate` directive (see the last chapter) The `release` directive does not physically free the HWA but marks it for reallocation.

If a release directive is used, this one must be executed last after all other instructions of the directive set.

If no group is defined, this directive is optional when the callsite is synchronous but is mandatory otherwise (like `delegatedstore`). The syntax of the directive is the following:
In stand-alone codelet context:

```
#pragma hmpp codelet_label release
```

In group of codelets context:

```
#pragma hmpp <grp_label> release
```

Where the directive parameters are:

- `<grp_label>` a unique identifier associated to all the directives that belong to the group (definition and use).
- `codelet_label`a unique identifier associated to all the directives that belong to the same codelet execution (definition and use).

Warning: Note that by default, if no group is defined, in case where a callsite is not associated to a release directive, the HWA is instantly released after the codelet execution has completed.

Listing 10 – release directive example (case of stand-alone codelet notation)

```
 ......
  while (j){
     for (k = 0 ; k < iter ; k++) {
#pragma hmpp testlabel1 callsite
       simplefunc1(n, &(t1[k*n]), &(t2[k*n]), &(t3[k*n]));
     }
    j--;
  }
#pragma hmpp testlabel1 release
 ......
```

Listing 10 shows a usage of the release directive. The allocated HWA of the testlabel1 call site is released after the while loop.

3.5 Controlling Data Transfers

When using an HWA, an important bottleneck is often the data transfer between the HWA memory and the host memory. To limit the communication overhead, the programmer can try to overlap data transfers with successive executions of the same codelets by using the asynchronous property of the HWA. Two directives can be used for that purpose:

- The `advancedload` directive loads data before the remote execution of the codelet.
- The `delegatedstore` directive delays the fetching of the result.

These directives are detailed in the next sections.

3.5.1 advancedload Directive

Data can be uploaded before the execution of the codelet by using the advancedload directive.

The syntax is:

In stand-alone codelet context:

```
#pragma hmpp codelet_label advancedload
                      ,args[arg_items]
                     [,args[arg_items].size={dimsize[,dimsize]*}]*
                     [,args[arg_items].addr="expr"]*
                     [,args[arg_items].section={[subscript_triplet,]+}]*
                     [,asynchronous]
```

In group of codelets context:

```
#pragma hmpp <grp_label> [codelet_label]? advancedload
                    ,args[arg_items]
                    [,args[arg_items].size={dimsize[,dimsize]*}]*
                    [,args[arg_items].addr="expr"]*
                    [,args[arg_items].section={[subscript_triplet,]+}]*
                    [,asynchronous]
```

Where the directive parameters are:

- `<grp_label>` a unique identifier associated with all the directives that belong to the group (definition and use).
- `codelet_label` a unique identifier associated with all the directives that belong to the same codelet execution (definition and use).
- `args[arg_items]` the name or rank (caller program) of the argument to be loaded.
- `args[arg_items].size={dimsize[,dimsize]*}` gives an alternate way to evaluate the size of non-scalar codelet arguments. Each `dimsize` provides the size for one dimension. This parameter may be used when the `callsite` specifies a size that is not known in the `advancedload` directive used.
- `args[arg_items].addr="expr"` expr is an expression that gives the address of the data to upload.
- `args[arg_items].section={[subscript_triplet,]+}*` indicates that only an array section will be transferred to the device. See chapter 3.5.3 for further details.
- `asynchronous` indicates that the transfer can be performed asynchronously, meaning that it is a non-blocking transfer.

The `advancedload` directive is used on data whose the `intent` status is `in` or `inout`. An error message is generated otherwise.

Listing 11 - advancedload directive example (case of stand-alone codelet notation)

```
#pragma hmpp matvec advancedload, args[inm], args[inm].size={n,m}
  ....
  while (...){
#pragma hmpp matvec callsite, args[inm].advancedload=true, &
#pragma hmpp matvec           args[inm].size={n+1,m+1}, &
#pragma hmpp matvec           asynchronous
    matvec(n, m, (inc+(k*n)), inm, (outv+(k*m)));
    ....
#pragma hmpp matvec synchronize
#pragma hmpp matvec delegatedstore, args[outv]
    if (...) {
      for (i=0; i<m; i++) {
        inm[...] = 0.1;
      } /* endfor */
#pragma hmpp matvec advancedload, args[inm]
    } /* endif */
  } /* endwhile */
```

An example of the `advancedload` directive is given in Listing 11 . The `advancedload` directive at line 15 loads the `inm` matrix after it has been modified and before the next call to the codelet.

Warning: The expression used to specify the size and address of the arguments can be evaluated only when the advancedload is used. However, most inconsistencies are likely to be detected at compile time. Listing 12 shows an illegal use of the advancedload directive where an error message will be issued by the compiler.

Listing 12 - Illegal use of the advancedload directive (the actual arguments of the codelet is not in the scope of the advancedload directive).

```
void foo_xxx(int* N, float* CA, float* CX, float* CY) {
  ...
  /* Illegal preloading of the "table" input data because
        table is declared below (“table” designated here as args [0]) */
#pragma hmpp callfoo advancedload, args[0], &
#pragma hmpp callfoo                  asynchronous
    ...
  /* Call the codelet */
    {
         float table[2];
         table[0] = 3.14159265357;
         table[1] = 2.718281;
  #pragma hmpp callfoo callsite, args[0].advancedload=true, &
  #pragma hmpp callfoo              asynchronous
         foo_hmpp(table, CX, CY, SY_out);
      }
      ...
  #pragma hmpp callfoo synchronize
      /* Starting from there, the codelet execution has complete */
      ...
  #pragma hmpp callfoo delegatedstore, args[SY_out]
      /* Starting from there, the value of SY_out has been updated */
      ...
  #pragma hmpp callfoo release
      /* Starting from there, the hardware can be reallocated
          to another codelet */
  }
```

When the execution reaches an `advancedload` program point, the HWA, if available, is locked by the ENZO™ runtime. When an asynchronous `advancedload` directive is used, the argument must not be modified between that directive and the call of the codelet.

3.5.2 delegatedstore Directive

The `delegatedstore` directive is the opposite of the `advancedload` directive in the sense that it downloads output data from the HWA to the host. The program execution is pause until all transfers are completed. The syntax is:

In stand-alone codelet context:

```
#pragma hmpp codelet_label delegatedstore
                      ,args[arg_items]
                      [,args[arg_items].addr="expr"]*
                      [,args[arg_items].section={[subscript_triplet,]+}]*
```

In group of codelets context:

```
#pragma hmpp <grp_label> [codelet_label]? delegatedstore
                      ,args[arg_items]
                     [,args[arg_items].addr="expr"]*
                     [,args[arg_items].section={[subscript_triplet,]+}]*
```

Where the directive parameters are:

- `<grp_label>` is a unique identifier associated with all the directives that belong to the group (definition and use).
- `codelet_label` is the unique identifier associated with all the directives that belong to the same codelet execution (definition and use);
- `args[arg_items4]` is the name (caller program) or rank of the codelet arguments to download.
- `args[arg_items].addr="expr"`: `expr` is an expression that gives the address of the data to store.
- `args[arg_items].section={[subscript_triplet,]+]*` indicates that only an array section will be transferred to the device. See chapter 3.5.3 for further details.

An example of the `delegatedstore` directive is given in Listing 13 . In this example, the simple function is called twice. Only the first call is a candidate for remote execution, so only that call is offloaded to an accelerator or a worker thread. The value of `myoutv1` is downloaded after the second call.

Note that for an asynchronous callsite a `delegatedstore` directive must be preceded by a `synchronize` directive.

The delegatedstore directive is used on data whose the `intent` status is `inout` or `out`. An error message is generated otherwise.

Listing 13 - delegatedstore directive example

```
#pragma hmpp simple callsite, asynchronous
  simple(n, m, myinc1,inm, myoutv1);
  simple(n, m, myinc2,inm, myoutv2);
#pragma hmpp simple synchronize
#pragma hmpp simple delegatedstore, args[outv]
#pragma hmpp simple release
```

Warnings:

- You have to ensure that the argument expression stays valid in the context of the delegatedstore use.
- This directive is mandatory in the context of asynchronous callsite.

3.5.3 Array Sections in HMPP

An array section is a selected portion of an array. It designates a set of elements from an array.

The array sections can be used in order to optimize data transfers between the host and the HWA in some cases where it is not necessary to transfer the whole array..

This parameter can be used with both the `advancedload` and the `delegatedstore` directives (see respectively chapter 3.5.1 and 3.5.2).

The syntax of this parameter is of the form:

```
args[arg_item].section={[subscript_triplet,]+}*
```

Where

- `arg_item` designates an array;
- `subscript_triplet` consists of two subscripts and a stride and defines a sequence of numbers corresponding to array element positions along a single dimension.

  The notation for the subscript_triplet is: “start:end:stride” where:
  - `start` and `end` are subscripts which designate the first and last values of a dimension.
  - `stride` is a scalar integer expression that specifies how many subscript positions to count to reach the next selected element. If the stride is omitted, it has a value of 1. The stride must be positive.

The `subscript_triplet` must be specified for each dimension of the array.

Warnings: Array sections must be used carefully in ENZO™ applications. Indeed, the use of a stride greater than 1 may results to a slowdown of the application when lots of data are transferred. In such cases, the transfer of the whole array still remains the best solution.

To get performance, users should not forget the constraints inherent in data layout:

- They should favor the transfer of contiguous data;

- They should favor data locality in array section (this means, for example, to transfer data by column for Fortran and by row for C and C++ instead of the opposite).

3.5.3.1 Case of not normalized arrays

By default the HMPP standard makes the assumption that the arrays are normalized, meaning that all the dimensions of the arrays:

- Start from 0, in C and C++;
- Start from 1, in Fortran;

In cases where at least one of an array's dimensions is not normalized, the shape must be specified using the following notation:

```
args[arg_item].section={[subscript_triplet,]+}* of {[shape_couple,]+}
```

Where `shape_couple`: designates the first and the last values in the sequence of indices for a dimension.

Listing 14 illustrates the approach. In the `delegatedstore` directive, the array section requests the transfer of the contiguous data `u[0:1024]` of a one dimension array `u` declared with the `(-1024:1024)` array shape.

Listing 14- array section specified with a shape (extract) (Fortran)

```
...
INTEGER, PARAMETER :: M=4
INTEGER, PARAMETER :: Ns=-1024
INTEGER, PARAMETER :: Ne=+1024
REAL :: u(Ns:Ne) , v(Ns:Ne)
...
!- Transfer of the whole array
!$HMPP <conv> advancedload, args[f1::A]
!- callsite
!$HMPP <conv> f1 callsite
call doubleconv1d(Ne-Ns,M,u,v,coef)
...
!- callsite
!$HMPP <conv> f2 callsite
call conv1d(Ne-Ns,M,u,coef)
...
!- get only the modified data on the host
!$HMPP <conv> delegatedstore, args[f1::A],args[f1::A].section={0:Ne} of { Ns:Ne }
.
.
.
!------------
! Codelet declaration
!--------------
!$HMPP <conv> f1 codelet
  SUBROUTINE doubleconv1d(n,iter,A,B,C)
.
.
.
```

3.5.3.2 Use of array sections in HMPP, examples

Below are a few examples provided to illustrate the use of the `.section` parameter.

Listing 15 - array section in advancedload directive - Transfer of 1 column (Fortran)

```
INTEGER, PARAMETER :: size = 3661
INTEGER*4, dimension(size,size) :: tab
...
  !$hmpp <Mygroup> get_col advancedload, args[tab], args[tab].section={1:size,1:1}
...
  !$hmpp <group> get_col callsite, args[tab].advancedload=true
  call put(size, tab)
...
```

On Listing 15, the user transfers the first column through the use of an `advancedload` directive, and on Listing 16 transfers the first row of the array `tab`. The a`dvancedload` parameter is set to true at the callsite level to notify that the transfer of the data has already been done.

Listing 16 - array section in advancedload directive - Transfer of 1 row (Fortran)

```
INTEGER, PARAMETER :: size = 3661
INTEGER*4, dimension(size,size) :: tab
...
  !$hmpp <Mygroup> get_col advancedload, args[tab], args[tab].section={1:1,1:size}
...
  !$hmpp <group> get_col callsite, args[tab].advancedload=true
  call put(size, tab)
...
```

3.6 HMPP data declaration

3.6.1 map directive

In a group, arguments from different codelets may share resources on the device. For example, they may refer to the same table or one may use the result of another one. In these cases, HMPP directives can take advantage of using the same memory space on the device for all these arguments.

The `map` directive provides this feature: it maps several arguments on the device.

The notation is the following:

```
#pragma hmpp <grp_label> map, args[arg_items]
```

The Listing 17 below illustrates the use of the `map` directive (in same color the “mapped” variables):

- Line 2: is the definition of a group of codelets;
- Line 3: illustrates the mapping of respectively two variables named `v1` defined in two different codelets names `init` and `dotSum`.
- Line 4: illustrates the mapping of respectively two variables named `lxp` and `v2` defined in two different codelets names `init` and `dotSum`.

From the HMPP point of view, the introduction of these two “map” directives means that:

- The two variables `v1` will be seen as the same on the device;
- The two variables `lxp` and `v2` will be seen as the same;

Warning: The IO status may be still different for each directive because they each refer to different callsites: this will determine the transfer requirements. However, the union set of IO directives will define the way that the map memory will be allocated!

Example: in a `map: a, b`

- `a` is `in` in one codelet.
- `b` is `out` in another codelet.
- The memory allocation will be `inout` (only one for both).
- `a` will be initialized before the first codelet.

- `b` will be downloaded after the second codelet.

Listing 17 - map directive example

```
...
#pragma hmpp <myGroup> group, target=TESLA1 // definition of the group
#pragma hmpp <myGroup> map, args[init::v1;dotSum::v1]
#pragma hmpp <myGroup> map, args[init::lxp;dotSum::v2]
#pragma hmpp <myGroup> init codelet, args[v1].io=out
void init(int n, float v1[n], float initval, float lxp[n]) {
  int j;
   for (j = 0 ; j < n ; j++)
     v1[j] = initval + lxp[j];
     ...
}
#pragma hmpp <myGroup> dotSum codelet, args[v1].io=inout
void dotSum(int n, float v1[n], float v2[n])
{
   int j;
   for (j = 0 ; j < n ; j++)
     v1[j] += v2[j];
}
```

To be able to be mapped, the variables must:

- Have the same dimensions
- Have the same type

The example given below shows an illegal map association between two array variables and a scalar. In such situations an error message will be generated.

Listing 18-Illegal map directive usage

```
...
#pragma hmpp <myGroup> group, target=TESLA1
#pragma hmpp <myGroup> map, args[dotSum::v1;init::n]
#pragma hmpp <myGroup> init codelet, args[v1].io=out
void init(int n, float v1[n]) {
    int j;
    float val = 0.0;
    for (j = 0 ; j < n ; j++)
      v1[j] = val++;
}
#pragma hmpp <myGroup> dotSum codelet, args[v1].io=inout
void dotSum(int n, float v1[n], float v2[n])
{
  int j;
  ...
}
```

3.6.2 The mapbyname Directive

This directive is quite similar to the `map` directive except that the arguments to be mapped are directly specified by their name. So, the notation is the following:

```
#pragma hmpp <grp_label> mapbyname [,variableName]+
```

To be able to be mapped, the same constraints as for the `map` directive apply, the variables must have:

- The same dimensions
- The same type

Listing 19 shows a use of this directive. In the group `<fxx_myGroup>` all of the variables called `xmin` will be mapped together, all of the named `xmax` will be mapped together, and so on.

Listing 19 - mapbyname directive example

```
!$hmpp <fxx_myGroup> mapbyname, xmin,xmax,ymin,ymax,zmin,zmax
```

The `mapbyname` directive is equivalent to multiple `map` directives.

```
!$hmpp <fxx_myGroup> mapbyname, xmin, xmax
```

Is equal to:

```
!$hmpp <fxx_myGroup> map, args[*::xmin]
!$hmpp <fxx_myGroup> map, args[*::xmax]
```

3.6.3 The resident directive

The `resident` directive declares some variables as global within a group. Those variables can then be directly accessed from any codelet belonging to the group. In practice, it means that those variables will reside in the HWA memory. So they can be seen as memory-resident variables on the HWA for the considered group.

This directive applies to the declaration statement just following it in the source code.

The syntax of this directive is:

```
#pragma hmpp <grp_label> resident
                         [, args[::var_name].io=[in|out|inout]]*
                         [, args[::var_name].size={dimsize[,dimsize]*}]*
                         [, args[::var_name].addr="expr"]*
                         [, args[::var_name].const=true]*
```

Where the directive parameters are:

- `<grp_label>` : a unique identifier associated to all the directives that belong to the group (definition and use).
- `args[::var_name].io=in|out|inout` indicates that the specified variables are either input, output or both. By default, unqualified variables are inputs. The specification of this parameter drives the data transfers between the host and the HWA. Furthermore, it allows some additional checks about the use of the data in ENZO™ applications (see chapter 3.4.1 for more details about the management of this property)..
- `args[::var_name].size={dimsize[,dimsize]*}` specifies the size of a non scalar parameter (an array). Each dimsize provides the size for one dimension. The set is evaluated at runtime by an allocate directive, or by all callsite and advancedload directives within the group.
- `args[::var_name].addr="expr"` expr is an expression that gives the address of the data to upload.
- `args[::var_name].const=true` indicates that the argument is to be uploaded only once. Note that even if there is only one callsite associated to a codelet declaration, there can be several calls to the codelet (when inserted inside a loop for instance). If a release directive is used between the calls, the data will be reloaded.

The notation `::var_name` with the prefix `::`, indicates an application's variable declared as resident.

Note that, unlike input or output codelet arguments, resident variables are never implicitly transferred to and from the HWA. Explicit advancedload and delegatedstore directives are required when necessary.

Listing 20 - resident directive example

```
#include <stdio.h>
#define SIZE 10240
// group declaration. The group label is “myGroup”
#pragma hmpp <myGroup> group, target=TESLA1
// resident data declaration inside the group "MyGroup"
#pragma hmpp <myGroup> resident, args[::tab_init_on_hwa].io=out &
#pragma hmpp <myGroup>           , args[::tab_init_on_host].io=in
float tab_init_on_hwa [SIZE], tab_init_on_host[SIZE];
// declaration of the codelet "init" inside the group "MyGroup"
#pragma hmpp <myGroup> init codelet
void init(int n) {
   int j;
   float val = 0.0;
   for (j = 0 ; j < n ; j++) tab_init_on_hwa[j] = val++ ;
}
// declaration of the codelet "dotSum" inside the group "MyGroup"
#pragma hmpp <myGroup> dotSum codelet
void dotSum(int n)
{
   int j;
   for (j = 0 ; j < n ; j++) tab_init_on_hwa[j] += tab_init_on_host[j];
}
int main(int argc, char **argv)
{
   int i, m=SIZE;
   float val = 0.0;
   for (i = 0 ; i < m ; i++)   tab_init_on_host[i] = val++*2;
#pragma hmpp <myGroup> allocate // allocation of the group on the HWA
// transfer onto the HWA of the variable tab_init_on_host
#pragma hmpp <myGroup> advancedload, args[::tab_init_on_host]
#pragma hmpp <myGroup> init callsite      // call to the "init" codelet
   init(m);
#pragma hmpp <myGroup> dotSum callsite     // call to the "dotSum" codelet
   dotSum(m);
  //transfer of the data from the HWA to the CPU
#pragma hmpp <myGroup> delegatedstore, args[::tab_init_on_hwa]
#pragma hmpp <myGroup> release // release of the HWA
   // short display of the results
   for (i = 0 ; i < m ; i=i+2) {
     if ((i <= 5) || (i >= m-5))
      printf ("tab_init_on_hwa[%d]= %4.2f \t\t tab_init_on_hwa[%d]= %4.2f \n",
              i, tab_init_on_hwa[i], i+1, tab_init_on_hwa[i+1]);
   }
   return 0;}
```

The Listing 20 illustrates the use of this directive. The corresponding results are presented on Listing 21

Listing 21 - Results of the application described Listing 10 (with hmpp and usual compiler like gcc)

```
$ pathcc MyProgramWithResident.c -o MyProgramWithResident
$ ./MyProgramWithResident
tab_init_on_hwa[0]= 0.00          tab_init_on_hwa[1]= 3.00
tab_init_on_hwa[2]= 6.00          tab_init_on_hwa[3]= 9.00
tab_init_on_hwa[4]= 12.00         tab_init_on_hwa[5]= 15.00
tab_init_on_hwa[10236]= 30708.00  tab_init_on_hwa[10237]= 30711.00
tab_init_on_hwa[10238]= 30714.00  tab_init_on_hwa[10239]= 30717.00
```

3.7 Regions in HMPP

This section presents a new set of HMPP directives to allow expressing computations to be performed on the HWA as regions of code. The goal is to avoid requiring code restructuring to build codelets.

A region is a merging of the codelet/callsite directives. Therefore, all of the attributes available for codelet or callsite directives can be used on regions directives.

In C, the region directive must be inserted immediately before a block.

In Fortran, the region and the corresponding endregion directives must be inserted around a part of executable code.

The constraints for writing regions are the same as for codelets (see chapter 2.1 for more details). In addition, the control flow must remain inside the region; that is, there must not be any:

- `return` (in C) and `stop` (in Fortran);
- no `break` and `continue` (in C), `cycle` and `exit` (in Fortran) to a loop enclosing the region;
- `goto` to jump inside or outside the region.

We distinguish two parts in the declaration of a region: one dedicated to the codelet parameters, the other dedicated to the callsite parameters. So, the syntax for the definition of a region is the following:

Be careful: Do not confuse an HMPP section, which refers to an array section (see chapter 3.5.3, Array sections in HMPP) with HMPP regions, which refer to a block of statements.

In C and C++

```
#pragma hmpp [<MyGroup>] [label] region

                              [, args[arg_items].io=[in|out|inout]]*
                              [, cond = "expr"]
     Codelet parameters       [, args[arg_items].const=true]*
                              [, target=target_name[:target_name]*]

                              [, args[arg_items].size={dimsize[,dimsize]*}]*
                              [, args[arg_items].advancedload=[true|false]]*
                              [, args[arg_items].addr="expr"]*
     Callsite parameters      [, args[arg_items].noupdate=true]*
                              [, asynchronous]?

                              [, private=[arg_items]]*
    {
                                 C BLOCK STATEMENTS
     }
```

In Fortran

```
!$hmpp [<MyGroup>] [label] region

                              [, args[arg_items].io=[in|out|inout]]*
                              [, cond = "expr"]
    Codelet parameters        [, args[arg_items].const=true]*
                              [, target=target_name[:target_name]*]

                              [, args[arg_items].size={dimsize[,dimsize]*}]*
                              [, args[arg_items].advancedload=[true|false]]*
                              [, args[arg_items].addr="expr"]*
     Callsite parameters      [, args[arg_items].noupdate=true]*
                              [, asynchronous]?

                              [, private=[arg_items]]*
                                 FORTRAN STATEMENTS

!$hmpp [<MyGroup>] [label] endregion
```

Where the directive parameters are:

- All the codelet parameters refer to parameters available for the codelet directive (see chapter 3.4.1 , codelet Directive)
- All the callsite parameters refer to parameters available for the callsite directive (see chapter 3.4.3 , callsite Directive);
- `private`: specifies the variables that should be re-declared to be only used in the region. Typically, this parameter applies for loop induction variables. The HMPP `private` keyword usage is identical to the OpenMP `private` keyword.

Warning: The HMPP standard assumes all variables used in a region must be explicitly mentioned through one of the region parameters. See examples below. By default, arguments of HMPP region have an INOUT status.

Listing 22 and Listing 23 show the use of the region directives for C language. Note that all the variables referenced in the C block statements are declared either through the use of their INPUT/OUTPUT status (“io” clause) or through the private keyword for the temporary variables.

Listing 22 - region in HMPP (C example)

```
.
.
.
  #pragma hmpp <MyGroup> MyRegionLabel region, args[a;n].io=in, &
  #pragma hmpp <MyGroup>        args[r].io=out, args[r;a].size={n}, &
  #pragma hmpp <MyGroup>        private=[s,c,i]
  { /* start HMPP region */
      for( i = 0; i < n; ++i ){
        s = sinf(a[i]);
        c = cosf(a[i]);
        r[i] = s*s + c*c;
    }
  } /* end HMPP region */
.
.
.
```

Listing 23 - region in HMPP (Fortran example)

```
!$hmpp <MyGroup> allocate
   .
   .
   .
  !$hmpp <MyGroup> MyRegionLabel region, args[n;a].io=in,&
  !$hmpp <MyGroup>         args[r].io=out, private=[i,sin_sq,cos_sq]
  do i = 1,n
     sin_sq = sin(a(i)) ** 2
     cos_sq = cos(a(i)) ** 2
     r(i) = sin_sq + cos_sq
  enddo
  !$hmpp <MyGroup> MyRegionLabel endregion
   .
   .
   .
  !$hmpp <MyGroup> release
   .
   .
   .
```

The following restrictions apply:

- Regions cannot be nested;
- Asynchronous regions must have at least a label;
- Only `hmppcg` directives are allowed inside the region.

Warning: In Fortran, all variables accessed in a region must have their declarations in the same compilation unit. That is, at the present time, you can not create a region where a variable is defined in an external module.

4 Supported Languages

The HMPP codelet generators do not handle the full language for C, C++, or Fortran. This restrictions aim at ensuring portability of the code on most HWAs (for example, allowing pointer arithmetic in C language would forbid generation of code for many hardware platforms) and also performance.

Moreover, it should be noted that in addition to the restrictions brought by the HMPP standard, HWAs may impose additional limitations. End-users should pay attention to the current limitations of the hardware accelerators that they want to use by consulting hardware manufacturer's website.

4.1 Input C and C++ Code

As mentioned above, the HMPP codelet generators do not handle the full C language. The HMPP codelet generators take C99 input code so the array size can be specified in the parameter declaration. The remainder of this section is organized as follows:

- Section 4.1.1 describes the valid C constructs for HMPP;
- Section 4.1.2 shows how codelet parameter data sizes are addressed by the HMPP codelet generator.

4.1.1 Supported C Language Constructs

In this section we describe the language constructs which are supported by the HMPP codelet generators. The codelet prototype is preferably in C99 style in which all array sizes are specified in the declaration (see Section 4.1.2). Typically a codelet code looks like:

Listing 24 – C codelet code example

```
void simplefunc(int n, float s1[1], float v2[n], float v3[n]){
  int i;
  float r = s1[0];
  for (i = 0 ; i < n ; i++) {
    r += v2[i] * v3[i];
  }
  s1[0] = r;
}
```

Below are the language constructs supported by the HMPP codelet generators. If a construct is not supported, the HMPP codelet generator issues an error message and no codelet implementation is produced.

- Atomic data types
  - `char`, `unsigned char`, `short`, `unsigned short`, `integer`, `long`, `long long`, `unsigned integer`, `unsigned long`, `unsigned long long`;
  - `float`, `double`, `complex`
- Data structures
  - Structure containing only scalar atomic fields.
  - Multidimensional arrays of structures.
- Language constructs
  - All arithmetic, shift and comparison operations.
  - for loops with simple induction variables. The following styles of for loops are supported:

```
for (i=lowbound ; i<highbound ; i++){...}
for (i=lowbound ; i<=highbound ; i++){...}
for (i=lowbound ; i<=highbound ; i = i+s){...}
```

    Where lowbound and highbound are invariant in the loop. The step value `s` is an integer constant. Furthermore, the induction variable `i` cannot be modified in the loop body.
  - Conditional statements `if() ... else …`
  - Array accesses with affine `A[a*i+b]` index expressions.
  - Calls to intrinsic (see Section 4.2.1.13 for the list of supported intrinsic) and functions.

The following constructs are not supported in a codelet:

- Pointer data accesses and pointer arithmetic.
- `switch` and `case` statements.
- Data structures containing arrays or structures of arrays
- Function pointers

Warning: Initialization of structure using C99 style is not supported

4.1.2 Parameter Passing Convention for C Codelets

To implement the communications between the host and HWAs, it is necessary to provide the HMPP API runtime with the size of the data to be transfered to/from the HWAs. Thus, this size must be explicitly specified in the codelet parameters. Listing 25 illustrates this.

Warning: By default, no aliasing is allowed between codelet parameters.

Listing 25 - Parameter data size passing using C99 for codelets

```
/* C99 syntax */
#pragma hmpp csmain codelet, args[a].io=in, &
#pragma hmpp csmain           args[b].io=in, &
#pragma hmpp csmain           args[r].io=out
void csmain(unsigned int S, float r[S], float a[S], float b[S]) {
  unsigned i;
  for (i=0 ; i<S ; i++){
    r[i] = b[i] / sqrt(a[i]);
  }
}
```

4.1.3 Inlined functions

HMPP supports the inlining of functions with the following restrictions:

- The definition of the inlined function must be available in the compilation scope of the codelet;
- The inlined function must not have any HMPP directives;
- The inlined function must not be recursive;
- The inlined function must not access global variables

4.2 Input Fortran Code

The HMPP codelet generators do not support the full Fortran language. The subset taken into account is similar to the C subset described in Chapter 4.1 . The remainder of this section is organized as follow:

- Section 4.2.1 describes the supported Fortran language constructs.
- Section 4.2.2 indicates how codelet parameter data sizes are addressed by the HMPP codelet generators.

4.2.1 Supported Fortran Language Constructs

In this section we describe the language constructs that are supported by the HMPP codelet generators. Typically a codelet code looks like:

Listing 26 – Fortran codelet code example

```
!$hmpp simple codelet, target=TESLA1
SUBROUTINE simple(n,m,inv,inm,outv)
  IMPLICIT NONE
  INTEGER, INTENT(IN) :: n,m
  REAL, INTENT(IN) :: inv(n)
  REAL, INTENT(IN) :: inm(m,n)
  REAL, INTENT(OUT) :: outv(m,n)
  INTEGER :: i,j
   DO j = 1,n
     DO i = 1,m
       outv(i,j) = inv(j) * inm(i,j)
     ENDDO
   ENDDO
END SUBROUTINE simple
```

The language constructs presented below are the ones supported by the Fortran HMPP codelet generators. If a construct is not supported, the code generator issues an error and no codelet is produced.

4.2.1.1 Explicit declaration in codelet

The “IMPLICIT NONE” statement is required in Fortran codelet. All variables must be explicitly declared in Fortran codelets.

4.2.1.2 Supported Data Types

The table below summarizes the scalar data types that are supported within the codelets and shows how they are interpreted.

Table 5- Supported Fortran data types

F77	F90	Default	Implementation
INTEGER*1	INTEGER(1)		8bit signed
INTEGER*2	INTEGER(2)		16bit signed
INTEGER*4	INTEGER(4)	INTEGER	32bit signed
INTEGER*8	INTEGER(8)		64bit signed
REAL*4	REAL(4)	REAL	IEEE754 32bit float
REAL*8	REAL(8)	DOUBLE PRECISION	IEEE754 64bit float
LOGICAL*1	LOGICAL(1)		8bit
LOGICAL*2	LOGICAL(2)		16bit
LOGICAL*4	LOGICAL(4)	LOGICAL	32bit
CHARACTER*1	CHARACTER(1)	CHARACTER	8bit

Current restrictions:

- The `KIND` of all types is hard-coded to the values used by most Fortran compilers. In the future, they will be configurable for each Fortran compiler,
- User defined types via the `TYPE` statements are not allowed,
- The `CHARACTER` type and the character constants are only allowed for `LEN=1`. Virtually no operation except comparison is allowed on characters so they are of limited usage except when passed as arguments to the codelet.

4.2.1.3 Declarations

Declarations can be provided using the old F77 or the new F90 form:

```
INTEGER    a,b ! F77 form
INTEGER :: c,d ! F90 form
```

The attribute DIMENSION can also be used to specify array shapes:

```
INTEGER :: A(10)
INTEGER,DIMENSION(10) :: B
```

4.2.1.4 Parameters

PARAMETER statements and attributes are supported for scalar objects only.

```
INTEGER, PARAMETER :: N=42
INTEGER M
PARAMETER ( M = 42 )
```

4.2.1.5 Inlined functions

The HMPP standard supports the inlining of functions with the same restrictions as for C language (see chapter 4.1.3).

4.2.1.6 Intrinsic functions

Intrinsic functions used in codelets must have been declared through the use of the INTRINSIC Fortran statement. The example below illustrates the use of intrinsic functions in Fortran codelets.

```
...
REAL(8),DIMENSION(N) :: V
real(8),dimension(N,N) :: Loc
INTEGER :: J
INTRINSIC :: LOG, COS, SIN
...
```

4.2.1.7 Other Type Attributes and Declarations

Most type attributes introduced by Fortran90 are currently not supported in codelets (`POINTER`, `VOLATILE`, `TARGET`, ...). A noticeable exception is `INTENT` which is in fact recommended for all codelet arguments.

`COMMON`, `EQUIVALENCE`, `BLOCKDATA` and all declaration statements that may create aliasing between variables are not allowed in codelets.

4.2.1.8 Arrays

Array bounds should be fully specified using constants or scalar integer arguments of the codelet.

Current restrictions:

- Scalar integer arguments used to specify an array bound shall not be modified within the codelet. Ideally, they should have the `INTENT(IN)` attribute,
- Scalar integer arguments used to specify an array bound must appear before that array in the argument list,
- For better performance, it is recommended to use a constant or a single variable for the lower bound.

Below is a typical example:

Listing 27 – Fortran array declaration in codelet

```
SUBROUTINE codelet(m,n,A,B,C)
  INTEGER, INTENT(IN)    :: m,n
  INTEGER, INTENT(INOUT) :: A(100), B(m,n), C(0:m*n-1)
  ...
END SUBROUTINE
```

The following forms of arrays are not allowed:

- Assumed-size arrays as in `A(*)` or `B(100,*)`
- Assumed-shape and deferred-shape arrays as in `A(:)` or `B(3:)`

Remark: an array of the form A(:m) is allowed since its lower bound is by default equal to one.

4.2.1.9 IF statements

The following forms of `IF` statements are supported:

- `IF…ENDIF` constructs optionally with `ELSE IF` and `ELSE`:

```
IF (A>B) THEN
  C = 1
ELSE IF (A<B) THEN
  C = -1
ELSE
  C = 0
ENDIF
```

- Logical `IF` statements:

```
IF (A==B) C=0
```

Current restrictions:

- `SELECT CASE` constructs are currently not supported.
- `GOTO`s are not supported as well as arithmetic `IF` statements that are in fact disguised `GOTO`s.

4.2.1.10 Loops

The following forms of loops are supported:

- `DO` statements with index, start, end and an optional step. The index and all 3 expressions shall be of type integer.
- `DO WHILE` statements;
- Standalone `DO` - so a potentially infinite loop.

A `DO` construct must be terminated by an `ENDDO` statement. The old F77 form using a termination label is not allowed. `EXIT` and `CYCLE` statements are allowed within `DO` constructs.

Current restrictions:

- The step, if any, must be a simple constant (such as 1 or -2).
- No loop name shall be specified to an `EXIT` or `CYCLE` statement. They are applied to the first outer loop.
- The computation of the number of iterations in a loop of the form (a) is assumed not to overflow when computed using the type of the index. In practice, e.g. for `INTEGER*4`, the number of iterations shall not be greater than 2^{31}-1 (2147483647).

4.2.1.11 Modules

The current HMPP standard brings a preliminary support of Fortran modules. The objective is to provide users with the most frequently used constructions used in Fortran applications. Thus, scalar `PARAMETER` variables of types `INTEGER`, `LOGICAL`, `REAL` and `COMPLEX` defined in modules can be directly used in HMPP codelets.

However, this first implementation mainly focuses on `INTEGER` parameters. Thus, the following operations are supported on `INTEGER` type only:

- Constant definitions. Evaluation of expressions is supported for the usual `INTEGER` arithmetic operators `+`, `-`, `*`, `/`.

```
MODULE foo
  INTEGER, PARAMETER :: N=24, M=5
  INTEGER, PARAMETER :: P= ((N+1)*(M-5))/(M+N)
END MODULE foo
```

- `INTEGER` comparison and `LOGICAL` operators (`.OR.`, `.AND.`, `.EQ.`, ...)

```
MODULE foo
  INTEGER, PARAMETER
  INTEGER, PARAMETER
  LOGICAL, PARAMETER
  LOGICAL, PARAMETER
  LOGICAL, PARAMETER
  LOGICAL, PARAMETER
END MODULE foo
```

- Intrinsic functions to query type kind information (`SELECTED_INT_KIND`, `SELECTED_REAL_KIND` and `KIND`)

```
MODULE foo
  INTEGER, PARAMETER :: INT4 = SELECTED_INT_KIND(4)
  INTEGER, PARAMETER :: INT10 = SELECTED_INT_KIND(10)
  INTEGER, PARAMETER :: INT14 = SELECTED_INT_KIND(14)
  INTEGER, PARAMETER :: FLOAT_4_7 = SELECTED_REAL_KIND(4,7)
  INTEGER, PARAMETER :: FLOAT_P10 = SELECTED_REAL_KIND(P=10)
  INTEGER, PARAMETER :: FLOAT_R20 = SELECTED_REAL_KIND(R=40)
  INTEGER, PARAMETER :: FLOAT = KIND(1.0E0)
  INTEGER, PARAMETER :: DOUBLE = KIND(1.0D0)
END MODULE foo
```

Because of the difficulty to ensure consistent rounding in floating point arithmetic, operations on `REAL` or `COMPLEX` data types are not yet supported. It is however possible to define parameters of `REAL` or `COMPLEX` types as long as their expressions only contain:

- `REAL` constant (e.g. `1.2`, `1.2D0`, `1.2_4`, `1.2_INT4`)
- `COMPLEX` constant
- Unary operator `-`
- Parentheses
- References to other parameters of the same type

`REAL` conversions whether they are implicit or explicit are not supported. In practice that means that the expression must be of the exact same type as the parameter. For instance, the example below is correct if we assume that the default `REAL` kind is 4:

```
REAL(4), PARAMETER :: X1 = 3.1415
REAL , PARAMETER :: X2 = 3.1415_4
```

However, the following equivalent declarations containing an implicit and an explicit cast to `REAL(8)` will not be able to be evaluated:

```
REAL(8), PARAMETER :: Y1 = 3.1415
REAL(8), PARAMETER :: Y2 = REAL(3.1415,kind=8)
```

In practice, one could write the declaration which is similar even though it is not semantically equivalent:

```
REAL(8), PARAMETER :: Y =3.1415_8
```

Note: Fortran module support will be improved in future releases, so some of these limitations will be removed in the future.

4.2.1.12 Operations

Arithmetic operations are currently limited to scalars. Support for arrays should be available in future releases. All native operators are supported:

- Arithmetic: `+ - / * **`;
- Comparison: `> < >= < == /=` (and their 'dotted' forms: `.GT. .LT.` and so on);
- Logical: `.NOT. .AND. .OR. .EQV. .NEQV.`

4.2.1.13 Function Calls

Table 6 – Supported Intrinsic functions

Name	Type	Semantic
ABS(x)	REAL*n or INTEGER*n	Absolute value
LOG(n)	REAL*n	Natural logarithmic
LOG10(n)	REAL*n	Base-10 logarithmic function
SQRT(n)	REAL*n	Square root
MIN(a,b,...)	REAL*n or INTEGER*n	Minimum
MAX(a,b,...)	REAL*n or INTEGER*n	Maximum
MOD(a,b)	INTEGER*n	a modulo b
EXP(a)	REAL*n	Base-E exponential
COS(a)	REAL*n	Cosine
SIN(a)	REAL*n	Sine
TAN(a)	REAL*n	Tangent
ACOS(a)	REAL*n	Arc-Cosine
ASIN(a)	REAL*n	Arc-Sine
ATAN(a)	REAL*n	Arc-Tangent
COSH(a)	REAL*n	Hyperbolic Cosine
SINH(a)	REAL*n	Hyperbolic Sine
TANH(a)	REAL*n	Hyperbolic Tangent
ACOSH(a)	REAL*n	Inverse Hyperbolic Cosine
ASINH(a)	REAL*n	Inverse Hyperbolic Sine
ATANH(a)	REAL*n	Inverse Hyperbolic Tangent
IAND(a,b)	INTEGER*n	Bitwise AND
IOR(a,b)	INTEGER*n	Bitwise OR
IEOR(a,b)	INTEGER*n	Bitwise Exclusive-OR
NOT(a)	INTEGER*n	Bitwise NOT
REAL(a)		Convert a to REAL
DBLE(a)		Convert a to DOUBLE PRECISION (i.e. REAL(8))
INT(a)		Convert a to INTEGER
INT1(a)		Convert a to INTEGER(1)
INT2(a)		Convert a to INTEGER(2)
INT4(a)		Convert a to INTEGER(4)
INT8(a)		Convert a to INTEGER(8)

Only calls to intrinsic functions listed below are supported. All arguments should be of scalar type.

Warning: In Fortran, local variables can be stored in global memory and be initialized at startup. Then they keep their value between function calls. This is not the case in codelets where variable declared locally are assumed to be strictly local (as in C).

4.2.2 Unsupported statements in codelet

The following statements are not supported in HMPP Fortran codelets:

- `WHERE`, `SELECT`, `CALL`, `FORALL`, `GOTO`, `USE`, `CONTAINS`, `INCLUDE`;
- I/O statements: `OPEN`, `CLOSE`, …
- Memory statements: `==>`, `ALLOCATE`
- Arithmetic if

4.2.3 Parameter Passing Convention for Fortran codelets

To implement the communication between the host and the HWAs, it is necessary to provide the ENZO™ runtime API with the size of the data to be transfered to/from the HWAs. This is performed using the Fortran syntax with the array bound specified as an expression of the codelet parameters as shown in the example presented in Section 4.2.1 . In other words, a parameter declaration such as `A(*)` is not supported. The `INTENT(IN|INOUT|OUT)` clause is mandatory.

4.2.4 Knowns limitations

- The HMPP `size=`, `addr=`, `cond=`, and `section=` parameters are not yet supported.
- A codelet procedure and its callsite must lie within the same module or external program unit. Thus, a codelet procedure may be a module procedure or an internal subroutine, but not an external subroutine.
- For a particular group or standalone codelet, the `advancedload` and `delegatedstore` directives must be able to access the same actual arguments as the `callsite` directive. Thus, these directives must all be in the same procedure, or the arguments must be available to all of them by host association or use association.

5 Compiling HMPP Applications

PathScale ENZO™ provides developers with HMPP standards compliant compilers in order to easily build ENZO™ applications. PathScale ENZO™ currently comes with HMPP Fortran compilers and in Q3 will include support for HMPP C and C++.

5.1 Overview

In terms of use, PathScale ENZO™ works the same as the EKOPath compilers. However, the paths diverge at the final code generation phase.

Compiling an ENZO™ program is as simple as using the traditional EKOPath pathf90, pathcc or pathCC compiler drivers.

5.2 Common Command Line Parameters

We strive to make the ENZO™ compiler as easy to use as possible, but for more details on compiler options please reference ENZO_cli_guide.pdf

6 Running HMPP Applications

ENZO™ applications using the ENZO™ runtime library work just as regular applications. No extra steps are required at run time, as long as the runtime library is available on the target system.

6.1 Launching the Application

HMPP programs using the ENZO™ runtime are launched just like regular programs

```
$ ./program
```

7 HMPP Codelet Generators

HMPP codelet generator directives are converted by PathScale ENZO™ compiler to an intermediate representation, which is lowered down the same compilation path as regular HMPP directives.

8 Improved code generation and performance

HMPPCG (HMPP Codelet Generator) extends the base set of directives to provide optimized code generation and mapping of input codelets into the target code.

Most of the transformations described in this part apply on loops. A loop is a syntactic language construction expressing the repetition of some statements.

In HMPP, a transformation can be applied on a loop if:

- It has a unique induction variable;
- The number of iterations must be computable at run-time before entering the loop.

`for` loops in the C language and `DO` loops in Fortran are supported.

To optimize the code generated by ENZO™, two main types of directives are used:

- Some specifying loop properties;
- Others mentioning transformations to be applied on the loops.

More directives will be provided in future versions of the HMPP standard.

Please note that ENZO™ does not check for the incorrect usage of the directives. Be aware that misuse of the HMPPCG directives may lead to undefined behaviour.

8.1 HMPPCG Directives Syntax

The general syntax of the directives is (respectively for C, C++ and Fortran) the following:

C and C++:

```
#pragma hmppcg [(target)]? directive_type [clause] [, clause]*
```

Fortran:

```
!$hmppcg[(target)]? directive_type [clause] [, clause]*
```

Where:

- `target`: allows to restrict the execution of the directive to a specific target.

For example, on Listing 30 the hmppcg permute transformation will be applied regardless of the considered hardware accelerator.

Listing 30 - hmppcg directive (basic example)

```
#pragma hmppcg permute j, i
  for (i = 1; i < M - 1; ++i) // 0
    {
      for (j = 1; j < N - 1; ++j) // 1
        {
          B[i][j] = c11 * A[i - 1][j - 1] + c12 * A[i][j] ;
        }
    }
```

Warning: Note that all the directives described in this part are introduced by using the hmppcg keyword and do not contain any labels. They are dedicated to codelet generation and apply only on the codelet source code that they just precede. They can only be used in codelets or regions.

8.2 Interpretation order of the HMPPCG directives

With the HMPP standard, several transformations can be applied, one after the other, on a loop nest. Two modes or directives scheduling are provided:

- One based on lexical order in the source code (default mode);
- One base on the evaluation of an `order` clause.

In the first case, the order in which the transformations are executed follows these steps:

- Step 1: search the first HMPPCG directive;
- Step 2: apply the source code transformation given by the directive (or ignore it if the target does not match);
- Step 3: go back to step 1 with the resulting code until no transformation remain to be applied.

Users must be careful about the order in which the directives are applied. The directives are successively evaluated and their execution is performed on the code resulting from the previous transformation.

Table 7 below illustrates this mode of operation with the following assumptions:

- "dN" means which directives to apply;
- A symbolic notation is used.

Table 7 – interpretation order of hmppcg directive: lexical order

```
d1 permute k, i, j
   loop i        // i index
d2 unroll 2 i index
    loop j       // j index
d3 unroll 3
      loop k     // k index
        s1
        s2
```

1 – Initial code. First the directive d1 is applied

```
d3 unroll 3
 loop k     // k index
   loop i   // i index
d2  unroll 2
    loop j  // j index
      s1
      s2
```

2 - The execution of d1 leads now to have d3 in first position. So, the directive d3 will be the next directive applied.

```
   loop k     // loop k is unrolled
    loop i          // i index
d2   unroll 2
     loop j         // j index
         s1
         s2
```

```
loop k    // loop k is
unrolled
  loop i    // i index
   loop j // loop j is
unrolled
         s1
         s2
```

3 - Then d3 is applied. The execution of d3 does not change the order of the directive. Loop k is unrolled. The directive d2 will then be the next directive applied.

4 - d2 is now applied. There is no more directives to be applied

Otherwise, another mode is possible through the use of the "order" clause available in certain directives. In this mode, the "order" clause forces the execution of the directives in the increasing order of "order" attributes.

If several directives have the same order value, then they are executed in lexical order. Table 8 illustrates the use of this clause (with the same assumptions as previously indicated).

Table 8 - interpretation order of hmppcg directive: use of the order clause

```
d1 permute j,i
   loop i
d2 unroll 2,order=1
     loop j
```

```
d1 permute j,i
   loop i
d2 unroll 2,order=1
    loop j
```

```
d3 unroll 3,order=0
       loop k
         s1
         s2
```

```
       loop k // loop k is
unrolled
          s1
          s2
...
```

1 – Initial code. Directive d3 is first applied since the order directive has the smallest value.

2 - The directive d3 has been applied. The directive d2 will be the next directive applied

```
d1 permute j,i
   loop i
    loop j   // loop j is unrolled
     loop k // loop k is unrolled
         s1
         s2
...
```

```
loop  j     //  loop  j  is
unrolled
  loop i
      loop  k  //  loop  k  is
unrolled
          s1
          s2
...
```

The directive d2 has been applied. Then the last directive to execute is d1

4 – Loops I and J have been permuted. All the directives have been applied.

Currently, ENZO™ does not apply more than 5 transformations consecutively on a same loop nest.

8.3 HMPPCG: Loop Properties

The directives described in this part allow specifying some properties on loops. These properties are then used by the HMPP generator in order to optimize the generated code.

8.3.1 HMPPCG parallel Directive

This directive has to be used when the codelet generator is not able to compute the parallel properties of complicated loops.

According to the nature of the considered target accelerator (vectorial or parallel), the use of this directive will lead to different schemes of codelet generation A parallel loop is declared using the following directive:

```
#pragma hmppcg parallel
                [, reduce (operator:var [, operator:var]* ) ]*
                [, private (var [, var]* ) ]*
```

Where:

- `reduce` specifies that in the considered loop, a reduction operation is performed (see chapter 8.3.1.1 below);
- `private` specifies that each loop iteration should have its own instance of the variable. A `private` variable is not initialized and the value is not maintained for use outside of the loop.

This directive applies on the loop it precedes.

8.3.1.1 HMPPCG parallel: the reduce clause

The `reduce` clause allows the user to indicate that one or several reductions are done in the loop. Indeed, without this clause, the parallel execution of a loop with such an operation could lead to a wrong result.

- `operator` specifies a reduction operator (see Table 9)
- `var` is the name of a scalar variable referenced in the loop;

The table below presents the list of allowed reduction operators in the reduce clause.

Table 9 - List of reduction operators defined in HMPP

Operators	Meaning
+	Addition
*	Multiplication
min	Minimum
max	Maximum
and .and. &&	Logical and
or .or. ||	Logical or
ixor ieor ^	bitwise exclusive or
ior |	bitwise inclusive or
iand &	bitwise and

Listing 31 - hmppcg parallel clause with reduction operations

```
#pragma hmppcg parallel, reduce (+:ssx,+:ssy)
  for ( i = 0; i < NK; i++)
  {
    if (qqprim2[i])
      {
        qq[qqprim[i]] += 1.0;
        ssx = ssx + qqprim3[i];
        ssy = ssy + qqprim4[i];
     }
}
```

Listing 31 illustrates the use of the hmppcg parallel directive with two addition (i.e. +) reduction operations.

Note that the use of this directive forces ENZO™ to consider the loop parallel independently of any analysis carried out and in some cases may create conflicts between the directive specified and the loop analysis. The table below summarizes such situations:

Results of ENZO™ loop-kinds analysis	HMPPCG pragma used	Results

Parallel	None	Loop is computed on hardware accelerator
	Parallel	Loop is computed on hardware accelerator
	Noparallel	Loop is computed on the CPU
	Parallel, reduce	Loop is computed on hardware accelerator
Sequential	None	Loop is computed on the CPU
	Parallel	Loop is computed on the hardware accelerator. A warning message mentions that this execution could lead to erroneous result
	Noparallel	Loop is computed on the CPU
	Parallel, reduce	Loop is computed on hardware accelerator
Parallel with reduction	None	Loop is computed on the CPU
	Parallel	Loop is computed on the hardware accelerator. A warning message mentions that this execution could lead to erroneous result
	Noparallel	Loop is computed on the CPU
	Parallel, reduce	Loop is computed on hardware accelerator (with a warning message if a reduction variable is not mentioned in the reduce clause)

8.3.2 Inhibiting Vectorization or Parallelization

A non-parallel loop (i.e. sequential) is declared using the following directive:

```
#pragma hmppcg noParallel
```

The following example shows a loop nest where the use of the HMPP directives allows guiding the code generation.

Listing 32 – noParallel and parallel directives

```
#pragma hmppcg noParallel
for (i=0; i < n; i++) {
  A[i][n] = B[i+1];
#pragma hmppcg parallel
  for (j=0; j < n; j++) {
    D[i][j] = A[i][j] * E[3][j];
  }
}
```

This directive proves to be useful to control the gridification process of loops on targets such as Tesla.

Note that this directive forces ENZO™ to consider the loop as sequential independently of any optimization analysis. Such a loop will be executed on the CPU.

This directive applies on the loop it precedes.

8.3.3 HMPPCG Grid blocksize directive

This directive controls the number of threads in a block for the “gridification” of a loop nest..

This pragma can be put anywhere inside a codelet and it applies to every loop nest following the pragma in lexical order. If no pragma is supplied, the default value is used (“32x4”).

The syntax is:

```
#pragma hmppcg grid blocksize “nxm”
```

Where:

- `n` and `m` are the new dimensions of blocks sizes within the grid.

For example, for NVIDIA® architecture, typical values are: `16x16`, `32x8`, `64x2`, `32x4`.

Note that the optimal value of the block size is dependent on the loop nest and on the targeted hardware.

Listing 33 - hmppcg grid blocksize directive. Example of use

```
//Loops will be gridified with the default value
for (i=0; i < n; i++) {
   for (j=0; j < n; j++) {
      ...
  }
}
...
#pragma hmppcg grid blocksize 8x8
//Loops will be gridified with 8x8 threads in a block
for (i=0; i < n; i++) {
   for (j=0; j < n; j++) {
      ...
  }
}
...
// still gridified with 8x8 threads in a block
for (i=0; i < n; i++) {
   for (j=0; j < n; j++) {
      ...
  }
}
```

After having been set, if the value of the block size needs to be change in a codelet, a new `hmppcg grid blocksize` *directive can be added in the codelet (see example below).*

8.3.4 HMPPCG accelerated context queries

Within as HMPP codelet or region, the `hmppcg set` directive provides a way to obtain information about the current accelerated context.

The general syntax of the directive is: C and C++ syntax:

```
#pragma hmppcg set <varname> = <query>(<arguments>)
```

Fortran syntax:

```
!$hmppcg set <varname> = <query>(<arguments>)
```

Where:

- `<varname>` is a scalar integer variable
- is one of the supported HMPPCG query intrinsics
- is a comma separated list of arguments (if the query intrinsic needs any).

Alternatively, the query intrinsic can be replaced by a single default integer constant

```
#pragma hmppcg set <varname> = <constant>
```

The semantic of the `hmppcg set` directive is that of a standard assignment of the specified variable for all the specified HMPP targets.

Listing 34 - Illustration of the hmppcg set directive

```
PROGRAM test
  integer :: x
  !$hmpp foo callsite
  CALL foo(x)
  IF (x==0) THEN
     PRINT *, "The fallback was executed"
  ELSE
     PRINT *, "The NVIDIA target was executed"
  END IF
CONTAINS
  !$hmpp foo codelet, target=TESLA1
  SUBROUTINE foo(status)
    IMPLICIT NONE
    INTEGER, INTENT(OUT) :: status
    status = 0
    !$hmppcg set status = 1
  END SUBROUTINE foo
END PROGRAM test
```

This behavior allows detecting dynamically whether the fallback is executed or not as shown by Listing 34

Listing 35 - Example of the hmppcg set directive used to detect which target is executed

```
PROGRAM test
  integer :: x
  !$hmpp foo callsite
  CALL foo(x)
  IF (x==0) THEN
     PRINT *, "The fallback was executed"
  ELSE IF (x==1) THEN
     PRINT *, "The Tesla target was executed"
  END IF
CONTAINS
  !$hmpp foo codelet, target=TESLA1
  SUBROUTINE foo(status)
    IMPLICIT NONE
    INTEGER, INTENT(OUT) :: status
    status = 0
    !$hmppcg(CUDA) set status = 1
  END SUBROUTINE foo
END PROGRAM test
```

Combined with the ability to restrict any HMPPCG directive to a specific target, the `set` *directive allows detecting dynamically which target is currently executed (see Listing 35).*

8.3.4.1 The GridSupport() query

The `GridSupport()` intrinsic returns `1` if the current HMPP target supports the concept of loop gridification (targets TESLA1,TESLA2,...) and `0` otherwise.

C and C++ syntax:

```
#pragma hmppcg set <varname> = GridSupport()
```

Fortran syntax:

```
!hmppcg set <varname> = GridSupport()
```

This query is typically used to detect whether an implementation using shared memory is possible in a codelet.

Listing 36 - GridSupport query example

```
!$hmpp jacobi codelet, target=TESLA1
SUBROUTINE jacobi(n,A,B)
  IMPLICIT NONE
  INTEGER, INTENT(IN) :: n
  INTEGER, INTENT(INOUT) :: A(n,n), B(n,n)
  INTEGER :: i,j
  INTEGER :: grid_support
  grid_support = 0
  !$hmppcg set grid_support = GridSupport()
  IF (grid_support==1) THEN
     ! Implement here a version using shared memory
     ...
  ELSE
     ! Implement here a version without shared memory
     ...
  ENDIF
END SUBROUTINE jacobi
```

8.3.4.2 The gridification queries

A set of query intrinsics is provided so information about the current gridified loop nest is available. Due to their nature, these queries should only be used within a gridified loop nest. They are not strictly forbidden outside such loops but their result would then be inconsistent.

Each of the gridification query intrinsics exists in 3 forms:

- The `X` and `Y` forms respectively refer to the internal and external gridified loop.
- The `XY` refers to a linearized view of the gridification.
- The last form takes reference to the gridified loop through their index variable which is given as argument.

The following gridification queries are currently supported:

- `BlockSizeX()`, `BlockSizeY()`, `BlockSizeXY()` and `BlockSize(index)` provide the block size as specified by the `hmppcg grid blocksize` directive.
- `RankInBlockX()`, `RankInBlockY()`, `RankInBlockXY()` and `RankInBlock(index)` provide the ranks of the current thread within the current block. Numbering starts from 0.
- `RankInGridX()`, `RankInGridY()`, `RankInGridXY()` and `RankInGrid(index)` provide the rank of the current thread in the complete gridification. Numbering also starts from 0.
- `BlockIdX()`, `BlockIdY()`, `BlockIdXY()` and `BlockId(index)` provide the rank of the block in the gridification. Numbering also starts from 0.
- `BlockCountX()`, `BlockCountY()`, `BlockCountXY()` and `BlockCount(index)` provide the number of blocks in the gridification.

Listing 37 - The XY linearization formulas

```
BlockSizeXY()   = BlockSizeX() * BlockSizeY()
RankInBlockXY() = BlockSizeX() * RankInBlockY() + RankInBlockX()
BlockIdXY()     = BlockCountX() * BlockIdY()    + BlockIdX()
BlockCountXY()  = BlockCountX() * BlockCountY()
RankInGridXY()  = BlockIdXY() * BlockSizeXY() + RankInBlockXY()
```

Remark: Those intrinsics are all computed using 32bits integers. This is sufficient given the limitations of the current GPUs architectures. The only exception is `RankInGridXY()` *which may overflow in 32bit integers for large problem sizes (e.g. a typical CUDA GPU may accept up to 64K*64K blocks of up to 1024 threads).*

8.3.5 HMPPCG gridification support

The "`hmppcg grid`" directive provides a set of functionalities related to the gridification process:

- `hmppcg grid shared` declares that a local scalar or array variable must be allocated in shared memory (i.e. all threads in the current gridified block have access to it). For arrays, their dimensions must be constant and known at compile time. The directive must be located within the gridified loop while the shared object must be declared outside that loop.
- `hmppcg grid barrier` introduces a synchronization barrier between all threads of the current gridified block. This is typically needed to avoid race conditions when accessing objects placed in shared memory. It is important to notice that most targets require ALL threads in the current block to honour the barrier. As a consequence, barriers should never be placed inside divergent conditional statements (i.e. not executed identically by all threads) and use of the `hmppcg grid unguarded` directive may be necessary to ensure that all threads of the block are alive.
- `hmppcg grid unguarded` removes the guard normaly used to 'kill' the unneeded threads in the last blocks of each dimension of gridification. Consider for example a 1D gridified loop of 1000 iterations and a block size of 64. Without an `hmppcg grid unguarded` directive, the last blocks should only execute the loop body for (1000 modulo 64 =) 40 out of its 64 threads. The remaining 24 threads must do nothing and so are considered as dead. For an unguarded gridification those dead threads would be executed thus increasing the effective number of iterations from 1000 to 1024. Using an unguarded gridification is like increasing manually the loop upper bound such that the number of iterations becomes a multiple of the block size. Unguarded gridification is usually needed when using the `hmppcg grid barrier` directive that requires all threads of the block to be alive. It should be noted that in most cases some guards must be manually reinserted to insure that the loop indexes remains in the expected ranges.

A typical gridified loop using barriers and shared memory looks like this:

```
!$hmppcg grid blocksize 64x1
!$hmppcg grid unguarded
!$hmppcg parallel
DO i=1,n
  IF (i<=n) THEN
     ... write to shared memory
  ENDIF
  !$hmppcg grid barrier
  IF (i<=n) THEN
     ... read from shared memory
  ENDIF
  !$hmppcg grid barrier
ENDDO
```

To get further details or examples about the used of the shared memory, see document [R9] , section 4.6 "Exploiting the Shared Memory".

8.3.6 HMPPCG constantmemory directive

NVIDIA® devices use several memory spaces, which have different characteristics that reflect their distinct usages in CUDA applications. These memory spaces include global, local, shared, texture, and registers (see [R4] and [R5] for more details).

The directive described here helps to improve the performance by allowing the use of the constant memory available on NVIDIA® architecture.

Access to this memory space from an ENZO™ application is possible by the introduction of the following directive in the codelet definition:

C and C++ syntax:

```
#pragma hmppcg constantmemory <param> [, <size>]?
```

Fortran syntax:

```
!$hmppcg constantmemory <param> [, <size>]?
```

With:

- `<param>` the codelet‟s parameter (array or scalar)
- `<size>` the size of the array (number of elements). When scalar variables are defined, the size is optional or must be equal to 1.

It should be noted that by default, scalar variables are automatically placed in constant memory.

When specifying a TESLA target ENZO™ allows using up to 2KB of constant memory.

This directive applies on codelet parameters. In Fortran application, it must be introduced before executable statements.

8.4 HMPPCG: loop transformations

Unlike the directives described earlier, those described in this part specify some transformation to apply on a loop. These transformations are applied before the final code generation. Their application can provide better performance by improving computation scheduling or data locality.

8.4.1 Permute transformation

The loop permutation is a common transformation which is usually used to improve data accesses locality but can also be used to create coarse-grain or fine-grain parallization.

This directive provides a way to permute nested loops. It may be very useful to reorder the loops according to the code that will be executed on CPU or on hardware accelerator. The order of loops may impact the coalescing of memory accesses.

The syntax is:

```
#pragma hmppcg permute <var>, <var> [, <var>]*
                    [, order = <order_value> ]?
```

Where:

- `<var>` identifies one of the loops, based on the name of its induction variable
- `<order_value>` is a positive number (starting at zero)

The application of this transformation reorganizes the loop control structures according to the new order specified by the directive.

Example:

Before

```
!$hmppcg permute k, i, j
DO I = 1, 8
  DO J = 1, 8
    DO K = 1, 8
      A(I, J, K) = B(I, J, K)*1.2
    ENDDO
  ENDDO
ENDDO
```

After

```
DO K = 1, N
    DO I = 1, N
      DO J = 1, N
        A(I, J, K) = B(I, J, K)*1.2
      ENDDO
    ENDDO
  ENDDO
```

The loops now follow the order k, I, j, as specified in the directive.

8.4.2 Distribute transformation

In some situations, loops may be too complex to be automatically parallelized:

- It may contain statements which prevent the parallelization

- The generated loop can use too many registers, which prevents effective execution

The distribute transformation splits the initial loop into several separate loops. This directive has two parts:

- The first part identifies the loop on which the transformation will be applied
- The second part identifies where the loop shall be cut off.

The syntax is:

```
#pragma hmppcg distribute
                          [, addtoall {<dir> [; <dir>]*} ]*
                          [, order = <order_value> ]?
#pragma hmppcg cut  [, add       {<dir> [; <dir>]*} ]*
```

Where:

- `<dir>` is a HMPP Codelet Generator directive. In this context, the hmppcg directive is written without the "language directive prefix".

  For example:

```
!$hmppcg distribute, addtoall {unroll 2, jam}
```

  Add an `hmppcg unroll` directive to the loops resulting from the application of the distribute clause.

- `<order_value>` is a positive number (starting at zero).
- The `addtoall` clause allows user to add new directives to the resulting loops created by the transformation.

The distribute directive is attached to the loop to be divided. This loop must contain at least one cut directive.

Listing 38 and Listing 39 illustrate the use of this transformation.

Listing 38 - Original code

```
DO I = 1, SIZE_1
!$hmppcg distribute
 DO J = 1, SIZE_2   ! original loop
 T(I, J) = 0
!$hmppcg cut
   DO K = 1, SIZE_2
     T(I, J) = A(I, K) * B(I, J)
   ENDDO
!$hmppcg cut
 C(I, J) = C(I, J) + T(I, J)
  ENDDO
 ENDDO
```

Listing 39 - Code after having applied the distribute transformation[1]

```
for (i_2 = 0, hmppcg_end = (*size_1) - 1; i_2 <= hmppcg_end; i_2 += 1)
 {
  for (j_21 = 0, hmppcg_end = (*size_2) - 1; j_21 <= hmppcg_end; j_21 += 1)
   {
    t[j_21][i_2] = 0;
   } // end loop j_21
  for (j_2 = 0, hmppcg_end = (*size_2) - 1; j_2 <= hmppcg_end; j_2 += 1)
    {
      for (k_2 = 0, hmppcg_end = (*size_2) - 1; k_2 <= hmppcg_end; k_2 += 1)
        {
          t[j_2][i_2] = (a[k_2][i_2]) * (b[j_2][i_2]);
        } // end loop k_2
    } // end loop j_2
  for (j_22 = 0, hmppcg_end = (*size_2) - 1; j_22 <= hmppcg_end; j_22 += 1)
    {
       c[j_22][i_2] = (c[j_22][i_2]) + (t[j_22][i_2]);
    } // end loop j_22
    } // end loop i_2
```

8.4.3 Fuse transformation

This transformation is the opposite of the previous one. If the granularity of a loop, or the work performed by a loop, is small, then the performance gain from its parrallization may be insignificant. This is because the overhead of parallel loop start-up is too high compared to the loop workload. In such situations, the `hmppcg fuse` transformation can be used to combine several loops into a single one, and thus increase the granularity of the loop.

To apply this transformation, the loops must have the same iteration space and must not be separated by any non-loops statements.

1 This is for educational purposes only since the real result differs from this presentation given here.

The syntax is:

```
#pragma hmppcg fuse <offset>
                            [, add {<dir> [; <dir>]*} ]*
                            [, order = <order_value> ]
```

Where:

- `<offset>` identifies the loops to consider. Value `0` designates the current loop (where the directive is set). So `-1` designates the first previous, `+1` the first next, `+2` the two next loops and so on
- `<dir>` is a HMPP Codelet Generator directive
- `<order_value>` is a positive number starting at zero.
- The add clause allows user to add new directives to the resulting loop created by the transformation.

Listing 40 to Listing 43 illustrate the use of this transformation.

Listing 40 - Original code

```
!$hmppcg fuse 1
 DO I = 1, N
    A(I) = B(I) - C(I)
 ENDDO
 DO J = 1, N
    IF(A(J) .LT. 0) A(J) = B(J)*B(J)
 ENDDO
```

Listing 41 - Code after having applied the fuse transformation[1]

```
for (i_2 = 0, __hmppcg_end = (*n) - 1; i_2 <= __hmppcg_end; i_2 += 1)
 {
   a[i_2] = b[i_2] - c[i_2];
   if (a[i_2] < 0)
   {
     a[i_2] = b[i_2] * b[i_2];
   }
 } // end loop i_2
```

Listing 42 - Original code –with negative fuse index

```
 DO I = 1, N
   A(I) = B(I) - C(I)
ENDDO
!$hmppcg fuse -1
DO J = 1, N
   IF(A(J) .LT. 0) A(J) = B(J)*B(J)
ENDDO
```

Listing 43 - Original code –with negative fuse index

```
for (i_2 = 0, hmppcg_end = (*n) - 1; i_2 <= hmppcg_end; i_2 += 1)
{
  a[i_2] = b[i_2] - c[i_2];
  if (a[i_2] < 0)
  {
    a[i_2] = b[i_2] * b[i_2];
  }
} // end loop i_2
```

8.4.4 Unroll directive transformation

1 This is for educational purposes only since the real result differs from this presentation given here

The loop unroll transformation is intended to increase register exploitation and decrease memory loads and stores per operation within an iteration of a nested loop. Improved register usage decreases the need for main memory accesses and allows better exploitation of some machine instructions.

This transformation can be applied by using the following directive:

```
#pragma hmppcg unroll { <var>:<factor> [, <var>:<factor>]* | <factor> [, <factor>]* }
                      [, remainder|noremainder|guarded [(<var> [, <var>]*)] ]*
                      [, contiguous|split|changestep   [(<var> [, <var>]*)] ]*
                      [, scalartemp|arraytemp]?
                      [, jam                           [(<var> [, <var>]*)] ]*
                      [, addtounrolled [(<var> [, <var>]*)] {<dir> [; <dir>]*} ]*
                      [, addtoremainder [(<var> [, <var>]*)] {<dir> [; <dir>]*} ]*
                      [, order = <order_value> ]
```

Where:

- `<var>` identify one of the loops, based on the name of its induction variable;
- `<factor>` is an unroll factor, strictly greater than zero (1 means no unroll performed, but the associated clauses are still executed.).
- The `addtounrolled` and `addtoremainder` clauses allow users to add new directives to the resulting loops created by the transformation.

Then the other clauses drive the loop unroll algorithm.

8.4.4.1 Dealing with the unroll strategy

Different schemas of unrolling can be used in the HMPP standard. These ones are controlled thanks to the following options:

- `contiguous`, which is the default behavior: the end bound is divided and arrays are accessed by a sequence of contiguous indexes.

Table 10 - unroll directive with contiguous option

Initial code

```
#pragma hmppcg unroll i:4, contiguous
for( i = 0 ; i < n ; i++ ) {
v1[i] = alpha * v2[i] + v1[i];
}
```

Extract of generated code (the remainder loop is not represented)

```
for (i_1 = 0, __hmppcg_end = (n / 4) - 1; i_1 <= __hmppcg_end;
  {
    v1[4 * i_1] = (alpha * (v2[4 * i_1])) + (v1[4 * i_1]);
    v1[(4 * i_1) + 1] = (alpha * (v2[(4 * i_1) + 1])) + (v1[(4 *
    v1[(4 * i_1) + 2] = (alpha * (v2[(4 * i_1) + 2])) + (v1[(4 *
```

```
    v1[(4 * i_1) + 3] = (alpha * (v2[(4 * i_1) + 3])) + (v1[(4 *
}
```

- `split`: array accesses are distributed along the iteration space.

Table 11 - unroll directive with split option

Initial code

```
#pragma hmppcg unroll i:4, split
   for( i = 0 ; i < n ; i++ ) {
        v1[i] = alpha * v2[i] + v1[i];
 }
```

Extract of generated code (the remainder loop is not represented)

```
for (i_1 = 0, __hmppcg_end = (n / 4) - 1; i_1 <= __hmppcg_end; i_1 += 1)
 {
   v1[i_1] = (alpha * (v2[i_1])) + (v1[i_1]);
   v1[i_1 + (n / 4)] = (alpha * (v2[i_1 + (n / 4)])) +
                       (v1[i_1 + (n / 4)]);
   v1[i_1 + ((n / 4) * 2)] = (alpha * (v2[i_1 + ((n / 4) * 2)])) +
                             (v1[i_1 + ((n / 4) * 2)]);
   v1[i_1 + ((n / 4) * 3)] = (alpha * (v2[i_1 + ((n / 4) * 3)])) +
                             (v1[i_1 + ((n / 4) * 3)]);
 }
```

- `changestep`: similar to contiguous, but the stride of the loop is multiplied by instead of recomputing accesses from the body of the loop. This strategy requires that the loop has no inter-iteration dependencies.

Table 12- unroll directive with changestep option

Initial code

```
#pragma hmppcg unroll i:4, changestep
  for( i = 0 ; i < n ; i++ ) {
       v1[i] = alpha * v2[i] + v1[i];
}
```

Extract of generated code (the remainder loop is not represented)

```
for (i_1 = 0, __hmppcg_end = ((n / 4) * 4) - 1; i_1 <= __hmppcg_end; i_1
+= 4)
    {
      v1[i_1] = (alpha * (v2[i_1])) + (v1[i_1]);
      v1[i_1 + 1] = (alpha * (v2[i_1 + 1])) + (v1[i_1 + 1]);
      v1[i_1 + 2] = (alpha * (v2[i_1 + 2])) + (v1[i_1 + 2]);
      v1[i_1 + 3] = (alpha * (v2[i_1 + 3])) + (v1[i_1 + 3]);
    }
```

8.4.4.2 Dealing with the remainder loop:

Like the unroll strategy, there are different ways to handle the remainder loop. The following keywords are provided:

- `remainder` is the default behavior. A remainder loop is generated when the number of iterations is unknown or if it is not modulo of the unrolling .
- `noremainder` can be used to prevent the generation of a remainder loop. This option must be used carefully. It forces ENZO™ not to generate a remainder loop (even when the number of iterations is not modulo of the unrolling factor).
- `guarded` is an alternate way to avoid the execution of a remainder loop by inserting guards inside the body of the loop unrolled.

8.4.4.3 Dealing with scalar variables

When applying a loop unroll and jam transformation, scalar variables can be handled in two ways:

- `scalartemp`, which is the default, temporary variables remain untouched. For example, for the loop nest containing the following statements and unrolled with a factor of two:

```
tmp += 1;
out[i1][i2] = in[i1][i2] + 1;
```

It will be transformed into:

```
tmp__0 = tmp__0 + 1;
tmp__1 = tmp__1 + 1;
out[2 * i1_1][i2_1__0] = (in[2 * i1_1][i2_1__0]) + 1;
out[(2 * i1_1) + 1][i2_1__0] = (in[(2 * i1_1) + 1][i2_1__0]) + 1;
```

- `arraytemp`: private variables accesses are transformed into an array. So in this context, a loop nest containing the following statements and unrolled on the first index with a factor of two and a jam:

```
tmp+=1;
out[i1][i2]=in[i1][i2]+1;
```

will be transformed into:

```
tmp[0] = (tmp[0]) + 1;
tmp[1] = (tmp[1]) + 1;
out[2 * i1_1][i2_11] = (in[2 * i1_1][i2_11]) + 1;
out[(2 * i1_1) + 1][i2_11] = (in[(2 * i1_1) + 1][i2_11]) + 1;
```

8.4.4.4 Jam clause

Finally, you can control the way duplicated statements are fused together:

- `jam [(<var> [, <var>]*)] ]*` enable the merge of duplicated child statements inside the specified loop

The `jam` argument designates a loop induction variable. The jam argument is optional.

By default, without any arguments, the `jam` clause applies to the most internal loop of the loop nest. If an argument is specified, this one specifies the loop in which the jam is applied.

The following examples given under the form of a pseudo-code to preserve the readability - illustrate the behavior of the `jam` clause:

- Table 13 : illustrates the default behavior of the `jam` clause. The loop is unrolled according to the loop induction variable, and then the structure of the loop nest is jammed.
- Table 14 : illustrates the use of the `jam` clause with an argument. The loop nest is not completely jammed according to the jam argument which specified that the jam must only be applied at the `i_loop` level (so only on the `j_loop`).

Table 13 - Illustration of the jam clause with no argument

Before

```
#unroll i:2, jam
loop i
  loop j
    loop k
     a(i,j,k)
    EndLoop k
  EndLoop j
EndLoop i
```

After transformation

```
loop i
  loop j
    loop k
       a(i,j,k)
       a(i+1,j,k)
    EndLoop k
  EndLoop j
EndLoop i
```

Table 14 - Illustration of the jam clause with argument (the k_loop is not jammed)

Before Intermediate state After transformation

```
#unroll i:2, jam(i)
loop i
  loop j
    loop k
      a(i,j,k)
    EndLoop k
  EndLoop j
EndLoop i
```

```
loop i
  loop j
    loop k
       a(i,j,k)
    EndLoop k
  EndLoop j
  loop j"
    loop k"
       a(i+1,j,k)
    EndLoop k"
  EndLoop j"
EndLoop i
```

```
loop i
  loop j
    loop k
       a(i,j,k)
     EndLoop k
    loop k
       a(i+1,j,k)
    EndLoop k
   EndLoop j
EndLoop i
```

Thus, on the original code below:

Listing 44 - Unroll and Jam transformation - Original code

```
#pragma hmppcg unroll i1:2, scalartemp, jam
  for(i1=0; i1<n1; i1++)
    {
      int tmp = 0;
      for(i2=0; i2<n2; i2++)
        {
          tmp+=1;
          out[i1][i2]=in[i1][i2]+1;
        }
    }
```

Listing 45 shows the results of the unroll transformation without the jam clause. The structure of the loop is duplicated two times

From the same initial code, Listing 46 shows the result obtained with the jam clause. Both loop control structures have been merged into a single one and the statements have been grouped together.

Listing 45 - Unroll transformation with no jam clause[1][2]

```
for (i1_1 = 0, _hmppcg_end = (n1 / 2) - 1; i1_1 <= _hmppcg_end; i1_1 += 1)
{
   tmp_0 = 0;
   {
     for (i2_1_0 = 0, _hmppcg_end = n2 - 1; i2_1_0 <= _hmppcg_end; i2_1_0 += 1)
      {
         tmp_0 = tmp_0 + 1;
         out[2 * i1_1][i2_1_0] = (in[2 * i1_1][i2_1_0]) + 1;
      }
   }
     tmp_1 = 0;
      {
      for (i2_1_1 = 0, _hmppcg_end = n2 - 1; i2_1_1 <= _hmppcg_end; i2_1_1 += 1)
       {
          tmp_1 = tmp_1 + 1;
          out[(2 * i1_1) + 1][i2_1_1] = (in[(2 * i1_1) + 1][i2_1_1]) + 1;
       }
     }
  }
...
```

Listing 46 - Unroll transformation with jam clause applied

```
for (i1_1 = 0, _hmppcg_end = (n1 / 2) - 1; i1_1 <= _hmppcg_end; i1_1 += 1)
{
  int32_t tmp_0;
  int32_t tmp_1;
  tmp_0 = 0;
  tmp_1 = 0;
  {
    int32_t _hmppcg_end, i2_1_0;
    for (i2_1_0 = 0, _hmppcg_end = n2 - 1; i2_1_0 <= _hmppcg_end; i2_1_0 += 1)
    {
      tmp_0 = tmp_0 + 1;
      tmp_1 = tmp_1 + 1;
      out[2 * i1_1][i2_1_0] = (in[2 * i1_1][i2_1_0]) + 1;
      out[(2 * i1_1) + 1][i2_1_0] = (in[(2 * i1_1) + 1][i2_1_0]) + 1;
    }
  }
}
```

1 This is for educational purposes only since the real result differs from this presentation given here
2 The remainder loop is not presented on this example

8.4.5 Full unroll transformation

This directive is used to fully unroll a loop and its nested loops. Fully unrolling a loop means that the loop is unrolled by its number of iterations and finally replaced by its body.

Of course, this directive can be applied provided that the number of iterations of all loops can be determined at compile-time (otherwise a transformation failure is issued).

The syntax is:

```
#pragma hmppcg fullunroll [<var>]
                          [, order = <order_value> ]
```

Where:

- `<var>` is the induction variable of the deepest nested loop which will be fully unrolled.
- `<order>` is a positive number starting at zero.

Listing 47 - fullunroll directive - original code

```
#pragma hmppcg fullunroll i1
  for(i1=0; i1<13; i1++)
    {
      #pragma hmppcg fullunroll i2
      for(i2=0; i2<10; i2++)
      {
        out[i1][i2]=in[i1][i2]+1;
      }
    }
```

Listing 48 – Code after applying the fullunroll transformation

```
{
  out(0, 0) = in(0, 0) + 1;
  out(0, 1) = in(0, 1) + 1;
  ...
  out(0, 9) = in(0, 9) + 1;
  out(1, 0) = in(1, 0) + 1;
  out(1, 1) = in(1, 1) + 1;
  out(1, 2) = in(1, 2) + 1;
  ...
  out(11, 9) = in(11, 9) +  1;
  out(12, 0) = in(12, 0) +  1;
  out(12, 1) = in(12, 1) +  1;
  out(12, 2) = in(12, 2) +  1;
  out(12, 3) = in(12, 3) +  1;
  out(12, 4) = in(12, 4) +  1;
  out(12, 5) = in(12, 5) +  1;
  out(12, 6) = in(12, 6) +  1;
  out(12, 7) = in(12, 7) +  1;
  out(12, 8) = in(12, 8) +  1;
  out(12, 9) = in(12, 9) +  1;
}
```

8.4.6 Tile transformation

This directive is used to divide the iteration space of perfectly nested loops into blocks. This transformation can improve the use of the memory hierarchy through the reuse of variables.

For each of the loops to tile:

- its iteration space is reduced to the wanted size;
- a new loop is created around to iterate between blocks

Applied to a set of loops, each newly created loop is placed outside the original set of loops. Original loops are not destroyed nor replaced. The table below sums up the transformation done:

Before After having applied the transformation

```
#pragma hmppcg tile i:2
 loop i
  loop j
   s1[i]
   s2[i]
```

```
loop i by 2
 loop i [1:2]
  loop j
   s1[i]
   s2[i]
```

The syntax is:

```
#pragma hmppcg tile { <var>:<size> [, <var>:<size>]*
                    | <size>       [, <size>]* }
       [, addtoouter [(<var> [, <var>]*)] {<dir> [; <dir>]*} ]*
       [, addtotiled [(<var> [, <var>]*)] {<dir> [; <dir>]*} ]*
       [, order = <order_value> ]
```

Where:

- <size> is the new value of one of the dimension of the iteration space of the loop nest
- <var> identifies a loop (based on its induction variable name.)
- <dir> is a HMPP Codelet Generator directive
- <order_value> is a positive number starting at zero.

Listing 49 and Listing 50 illustrate a simple example use of this transformation

Listing 49 - HMPPCG Tile transformation

```
#pragma hmppcg tile i:8
  for( i = 0 ; i < n ; i++ ) {
       v1[i] = alpha * v2[i] + v1[i];
  }
```

Listing 50 - code after having applied the HMPPCG Tile transformation

```
hmppcg_end_outer = (n - 1) / 8;
for (outer_i_2 = 0; outer_i_2 <= hmppcg_end_outer; outer_i_2 += 1)
{
  hmppcg_end_i_2 = ((((outer_i_2 * 8) + 7)>(n - 1)?
                          (n - 1) :
                          ((outer_i_2 * 8) + 7))) - (outer_i_2 * 8);
     for (i_2 = 0 ; i_2 <= hmppcg_end_i_2; i_2 += 1)
      {
        v1[i_2 + (outer_i_2 * 8)] = (alpha * (v2[i_2 + (outer_i_2 * 8)])) +
                                    (v1[i_2 + (outer_i_2 * 8)]);
      }
   }
}
```

9 Going further: factorization of the HMPP directives

ENZO™ provides a preprocessor which allows the programmer to factorize the declarations of HMPP directives.

The main purposes of having a preprocessor are:

- To simplify the writing of HMPP directives;
- To allow HMPP directives to be configured via compilation options.

The HMPP preprocessor will be run before the native language preprocessor, if any. In practice, it means that using the preprocessor features within included files (e.g. by a Fortran INCLUDE statement or a C `#include` directive) will not be possible.

The HMPP preprocessor is mostly inspired from the standard C preprocessor

9.1 General Rules for Preprocessor Commands

Preprocessor commands are directives similar to the HMPP directives. All preprocessor commands will start with character `#`, to distinguish them from other HMPP directives.

The general syntax for the HMPP preprocessor commands is in Fortran:

```
!$hmpp #KEYWORD [ARGUMENTS...]
```

and in C and C++:

```
#pragma hmpp #KEYWORD [ARGUMENTS...]
```

9.1.1 Display Commands

The display commands simply print their arguments.

Syntax:

```
!$hmpp #echo args
```

```
!$hmpp #error args
```

```
!$hmpp #warning args
```

Potential macros in arguments are expanded.

Arguments of #echo are printed to the standard output stream. Arguments of `#error` and `#warning` are printed to the standard error stream, prefixed with the location of the command.

An `#error` immediately stops the preprocessing and produces an error code, a `#warning` does not.

Note that the #echo command is mostly intented for debug and should not appear in release code.

9.1.2 #PRINT Command

The `#print` command allows printing the arguments into the output source file.
Syntax:

```
!$hmpp #print args
```

Potential macros in arguments are expanded.

9.1.3 #DEFINE Command without Argument

The `#define` command associates an arbitrary value to a symbolic name.

Syntax:

```
!$hmpp #define name value
```

The name can be any valid identifier. In the resulting code, the `#define` command is expanded to a single empty line.

After a `#define` command, each occurrence of \name or \{name} in a HMPP or HMPPCG directive is replaced by the specified value.

The following rules are applied during the definition of a macro:

- The first blank character (space or TAB) after “name” is not part of the value.
- The trailing newline character is not part of the value.
- In all directives, the characters \ can be escaped by doubling them as in \\.
- No expansion is performed on the value before affecting the macro.
- A `##` sequence indicates that the tokens on the left and right must be concatenated, ignoring all neighboring spaces (same semantic as in CPP).

Example 1: A simple macro usage

```
!$hmpp #define NB 4
!$hmppcg unroll(\NB), noremainder
```

Becomes:

```
!$hmppcg unroll(4), noremainder
```

Example 2: In this example, the `\X1` and `\X2` are both extended to `B` because the `\ARG` in their value is expanded during the `callsite` and not during the `#define` statements.

```
!$hmpp #define ARG A
!$hmpp #define X1 \ARG
!$hmpp #define ARG B
!$hmpp #define X2 \ARG
...
!$hmpp Foo callsite , args[\X1;\X2].noupdate
```

Becomes:

```
…
!$hmppc Foo callsite , args[B;B].noupdate
```

9.1.4 #DEFINE Command with Arguments

A macro can be specified with a list of arguments as follows:

Syntax:

```
!$hmpp #define name(arg [, arg ,...]) value
```

For each of the specified arguments, macros of that name can be expanded in the value.

The expansion follows the following rules:

- The argument hides any macro of the same that may exist in the expansion context.
- The argument is only visible during the first level of expansion of value (see the example below)
- The arguments are expanded before the macro;
- Commas ',' and closing parenthesis ')' characters are not allowed in the arguments before their expansion.

Example 1:

```
!$hmpp #define FOO(a,b) From \a to \b
!$hmpp #echo FOO(100,200)
```

Becomes:

```
From 100 to 200
```

9.1.5 #BLOCK and #INSERT without Arguments

The `#block` command marks the start of a named block of text. The block ends with the corresponding `#endblock` command.

A block defined can be later inserted using a `#insert` command

Syntax:

```
!$hmpp #block name
  body
!$hmpp #endblock name
  ...
!$hmpp #insert name
```

The body of the block is arbitrary. It is not interpreted in any way when the block is defined.

When the `#insert` directive is encountered, the lines forming the body are inserted and processed according to the usual rules.

Example: A block can be inserted in multiple places

```
!$hmpp #block MyLoad
!$hmpp <MyGroup> MyCodelet1 advancedload, args[A;B;C]
!$hmpp <MyGroup> MyCodelet2 advancedload, args[A;B;C]
!$hmpp #endblock MyLoad
IF (debug) THEN
  !$hmpp #insert MyLoad
ELSE
  PRINT *,'Begin Load'
  !$hmpp #insert MyLoad
  PRINT *,'End Load'
ENDIF
```

Becomes:

```
IF (debug) THEN
  !$hmpp <MyGroup> MyCodelet1
  !$hmpp <MyGroup> MyCodelet2
ELSE
  PRINT *,'Begin Load'
  !$hmpp <MyGroup> MyCodelet1
  !$hmpp <MyGroup> MyCodelet2
  PRINT *,'End Load'
ENDIF
```

9.1.6 #BLOCK and #INSERT with Arguments

Blocks can be defined with arguments.

Syntax:

```
!$hmpp #block name(arg1 [, arg2 ,...])
  body
!$hmpp #endblock name
...
!$hmpp #insert name(val1 [, val2 ,...])
```

The `arg…argN` are identifiers.
The `val1…valN` are arbitrary expressions with the following restrictions:

- They cannot contain commas ','
- They cannot contain closing parenthesis ')"

The rules for processing the arguments in a #insert directive are:

- A macro is defined for each `arg1…argN` using the corresponding `val1, …, valn`.
- A macro expansion is applied to `val1…valn` before affecting `arg1, …, argN`.
- After that expansion `val1…valn` are allowed to contain commas and closing parenthesis.
- The definition of `arg1…argN` is valid for the whole inserted body.

- The macros `arg1…argN` are restored to their original value after the `#insert`.
- `#define` or `#undef` to `arg1…argN` are only valid within the `#insert` (according to rule just before)
- `#define` or `#undef` applied to any other macros remain valid after the `#insert`

Example: A simple `#block` and `#insert` with arguments

```
!$hmpp #block myBlock(A,B)
!$hmpp #echo I say \B \A
!$hmpp #echo Oops! I say \A \B
!$hmpp #endblock myBlock
!$hmpp #insert myBlock(Hello,World)
```

Becomes:

```
I say World Hello
Oops! I say Hello World
```

10 ENZO™ Supported HWA

10.1.1 Hardware Accelerators

PathScale ENZO™ supports target=TESLA1 which provides support for the Tesla C1060 and C1070 cards.

To maintain compatibility with CAPS' HMPP compiler we currently alias target=CUDA to TESLA1. There is a compiler switch which allows this behavior to be overridden and users are encouraged to take advantage of this since this alias is considered unstable and may change with future versions of the compiler.

Glossary

callsite	In HMPP context, designates a codelet call in the application
Codelet	A routine to be remotely executed in an HWA. A codelet is a pure function. It is a small self-contained subset section of executable code whose dynamic execution consumes a significant amount of time
CUDA	Programming language for the NVIDIA® CUDA compatible hardware
Device	A particular HWA device
Guards	Predicates expressed using HMPP directives to define runtime conditions to execute a codelet RPC in an HWA
Hardware Accelerators (HWA)	A device used to speedup segments of an application. Typical examples of a an HWA are : GPU, FPGA, or streaming units (SSE, ...).
HMPP	A set of directives made an open standard by CAPS Entreprise

	and PathScale
HMPP codelet	Contains a pure function that can be executed in an HWA using HMPP. The HMPP codelet also contains the ENZO™ runtime callbacks
HMPP Group of codelets	A group of codelets designates the execution of several codelets based on a same hardware allocation and with the possibility to share data.
HMPP directives	Set of directives to program the use of HWAs in application source
HMPP native codelet	HMPP native codelet is the original function that is annotated using the HMPP directives
HMPP preprocessor	The HMPP preprocessor translates the HMPP directives into calls to the HMPP runtime library
ENZO™ program	A C, C++ or Fortran program that contains HMPP directives
HMPP region	A set of contiguous statements to be executed on the HWA.
ENZO™ runtime API	Runtime library linked with the ENZO™ program to manage the execution of the HMPP codelet.
ENZO™ runtime callbacks	API that provides the ENZO™ runtime with all the necessary services to execute a target codelet
HMPP target codelet	HMPP target codelet is the hardware dedicated implementation of the codelet
Label	A label identifying a group of directives defining the declaration and execution of a codelet.
main thread	Process that executes the original code
Remote Procedure Call (RPC)	In HMPP, an RPC denotes the remote execution of a codelet in an HWA
Resident variable	A resident variable points out a data of the program which can explicitly be declared at HMPP level as: • “global” for a group: means that this variable will be accessible from any codelets belonging to the considered group; • “local” on the HWA: means that once this variable has been loaded on the HWA, it stay available up to the release of the device. This kind of variable is introduced by the directive keyword “resident” (see chapter 3.6.3 for more details on this directive). The management of this kind of data (load to the HWA or write to the host) must be explicitly done at user level by using the “advancedload” and “delegatedstore” directives.

Bibliography

R1: , HMPP Workbench User Guide, 2010
R9: , HMPP – NVIDIA GPU FORTRAN and C Cookbook, Version 2.0, 2010
R4: , NVIDIA_CUDA_Programming_Guide_[2.1;2.2;2.3].pdf,
R5: , NVIDIA_CUDA_BestPracticesGuide_[2.3].pdf,